Trabajo Integrador - Programación I

Datos Generales

- **Título del trabajo:** Estructuras de Datos Avanzadas: Aplicación de Árboles Binarios en Python para la Gestión de Datos Jerárquicos.
- **Alumnos:**
  - Ignacio Malatesta
  - Pablo Limardo
- **Materia:** Programación I
- **Fecha de Entrega:** 10/06/2025

Índice

1. Introducción

En el campo del desarrollo de software, la eficiencia en la manipulación y organización de los datos es un pilar fundamental. A medida que los sistemas crecen en complejidad, las estructuras de datos lineales como listas o arreglos resultan insuficientes para representar relaciones complejas. Este trabajo se enfoca en las **estructuras de datos no lineales**, específicamente en los **árboles**, debido a su capacidad para modelar jerarquías de manera natural y eficiente.

La elección de este tema surge de nuestro interés por comprender cómo resolver problemas de almacenamiento y recuperación de información que van más allá de una secuencia simple. Los árboles están presentes en innumerables aplicaciones cotidianas, desde la estructura de carpetas de un sistema operativo hasta los índices de las bases de datos que agilizan nuestras búsquedas en la web. Su relevancia en la programación moderna es, por lo tanto, indiscutible.

El objetivo principal de este trabajo es doble. Primero, buscamos consolidar nuestra

comprensión teórica sobre los árboles, explorando sus distintas clasificaciones, propiedades y los algoritmos de recorrido fundamentales. Segundo, nos proponemos aplicar estos conceptos en un **caso práctico desarrollado en Python**, demostrando cómo una implementación de un árbol de búsqueda binaria puede optimizar la gestión de un conjunto de datos jerárquico, superando las limitaciones de los métodos tradicionales.

A través de esta investigación, aspiramos no solo a cumplir con los requisitos académicos de la materia Programación I, sino también a desarrollar una herramienta conceptual y práctica que nos servirá como base para futuros desafíos en el ámbito de la programación.

2. Marco Teórico

2.1 Conceptos Fundamentales de Árboles

Un **árbol** es una estructura de datos no lineal que simula una jerarquía mediante nodos conectados. A diferencia de listas o arreglos, los árboles permiten representar relaciones padre-hijo entre elementos. Cada árbol posee un **nodo raíz**, desde el cual se ramifican los **nodos hijos**. Aquellos nodos que no tienen hijos se denominan **hojas**. Las propiedades básicas de un árbol incluyen: un solo nodo raíz, cada nodo puede tener múltiples hijos, los nodos están conectados por **aristas** y no existen ciclos.

- **Longitud y Profundidad:** La longitud de un camino es el número de aristas entre dos nodos. La altura del árbol es la longitud del camino más largo desde la raíz a una hoja.
- **Nivel y Altura:** El nivel de un nodo es la longitud del camino que lo conecta a la raíz más uno. La altura de un árbol es el máximo nivel del mismo.
- **Grado y Orden:** El grado de un nodo es el número de hijos que tiene. El grado de un árbol es el grado máximo de sus nodos. El orden es la máxima cantidad de hijos que puede tener cada nodo, establecido como restricción.
- **Peso:** Es el número total de nodos que tiene un árbol, lo que da una idea de su tamaño en memoria.

2.2 Tipos de Árboles

Existen múltiples tipos de árboles, cada uno optimizado para diferentes escenarios.

- **Árbol Binario:** Es una estructura donde cada nodo tiene, como máximo, dos hijos (izquierdo y derecho). Es la base para estructuras más complejas y se utiliza en la representación de expresiones aritméticas.
- **Árbol de Búsqueda Binaria (BST):** Es un árbol binario ordenado que cumple una propiedad clave: los valores del subárbol izquierdo de un nodo son menores que

el valor del nodo, y los del subárbol derecho son mayores. Esto permite búsquedas, inserciones y eliminaciones muy eficientes.
- **Árbol N-ario:** Generalización del árbol binario donde un nodo puede tener hasta 'n' hijos. Son flexibles y se usan para representar jerarquías generales, como sistemas de archivos.
- **Árbol de Decisión:** Modelo utilizado en inteligencia artificial y aprendizaje automático. Cada nodo interno representa una prueba sobre un atributo, cada rama una salida de la prueba y cada hoja una decisión o clasificación final.

2.3 Recorridos de Árboles

Para procesar la información contenida en un árbol, es necesario "visitar" cada uno de sus nodos de manera sistemática.

- **Preorden (Raíz, Izquierda, Derecha):** Se visita primero la raíz, luego se recorre recursivamente todo el subárbol izquierdo y, finalmente, el derecho.
- **Inorden (Izquierda, Raíz, Derecha):** Se recorre recursivamente el subárbol izquierdo, se visita la raíz y, por último, se recorre el derecho. En un BST, este recorrido recupera los elementos en orden ascendente.
- **Postorden (Izquierda, Derecha, Raíz):** Se recorren recursivamente los subárboles izquierdo y derecho antes de visitar la raíz. Es útil para eliminar un árbol de la memoria.

3. Caso Práctico

Descripción del Problema

Una tienda de componentes electrónicos necesita un sistema para administrar su inventario. Los requisitos son: almacenar componentes por un SKU (ID numérico) único, agregar nuevos componentes, buscar rápidamente por SKU, eliminar componentes discontinuados y mostrar el inventario ordenado por SKU. Un Árbol de Búsqueda Binaria (BST) es la estructura ideal para este problema.

Código Fuente Comentado

```
#
=================================================================
=========
# DEFINICIÓN DE LAS CLASES
#
=================================================================
```

```
=========

class Nodo:
    """
    Representa un componente en el inventario.
    Cada nodo contiene los datos del componente y las referencias a
sus hijos.
    """
    def __init__(self, sku, nombre, stock):
        self.sku = sku
        self.nombre = nombre
        self.stock = stock
        self.izquierda = None  # Almacenará el sub-árbol izquierdo
(nodos con SKU menor)
        self.derecha = None    # Almacenará el sub-árbol derecho
(nodos con SKU mayor)

    def __str__(self):
        """Devuelve una representación en string del componente para
imprimirlo fácilmente."""
        return f"SKU: {self.sku}, Nombre: {self.nombre}, Stock:
{self.stock}"

class ArbolInventario:
    """
    Implementación de un Árbol de Búsqueda Binaria para gestionar el
inventario.
    Esta clase orquesta todas las operaciones sobre los nodos.
    """
    def __init__(self):
        # La raíz es el punto de partida del árbol. Inicialmente,
está vacía.
        self.raiz = None

    def agregar_componente(self, sku, nombre, stock):
        """
        Método público para agregar un nuevo componente.
        Este es el método que se llama desde fuera de la clase.
```

```
        """
        # Si el árbol está vacío, el nuevo nodo se convierte en la
raíz.
        if self.raiz is None:
            self.raiz = Nodo(sku, nombre, stock)
        # Si el árbol ya tiene nodos, usamos un método recursivo para
encontrar la posición correcta.
        else:
            self._agregar_recursivo(self.raiz, sku, nombre, stock)

    def _agregar_recursivo(self, nodo_actual, sku, nombre, stock):
        """
        Método privado y recursivo para insertar un nodo.
        Navega por el árbol hasta encontrar un lugar vacío donde
colocar el nuevo nodo.
        """
        # Comparamos el SKU del nuevo componente con el del nodo
actual.
        if sku < nodo_actual.sku:
            # Si es menor, vamos hacia la izquierda.
            if nodo_actual.izquierda is None:
                # Si no hay nada a la izquierda, hemos encontrado el
lugar. Lo insertamos aquí. (Caso base)
                nodo_actual.izquierda = Nodo(sku, nombre, stock)
            else:
                # Si ya hay un nodo, seguimos bajando por la rama
izquierda. (Paso recursivo)
                self._agregar_recursivo(nodo_actual.izquierda, sku,
nombre, stock)
        elif sku > nodo_actual.sku:
            # Si es mayor, vamos hacia la derecha.
            if nodo_actual.derecha is None:
                # Si no hay nada a la derecha, lo insertamos aquí.
(Caso base)
                nodo_actual.derecha = Nodo(sku, nombre, stock)
            else:
                # Si ya hay un nodo, seguimos bajando por la rama
derecha. (Paso recursivo)
```

```
                self._agregar_recursivo(nodo_actual.derecha, sku,
nombre, stock)
        else:
            # Si el SKU ya existe, no lo agregamos para evitar
duplicados.
            print(f"-> ADVERTENCIA: El componente con SKU {sku} ya
existe. No se puede duplicar.")

    def buscar_componente(self, sku):
        """Método público para buscar un componente por su SKU."""
        return self._buscar_recursivo(self.raiz, sku)

    def _buscar_recursivo(self, nodo_actual, sku):
        """
        Método privado y recursivo para encontrar un nodo.
        Aprovecha la propiedad del BST para descartar la mitad del
árbol en cada paso.
        """
        # Casos base de la recursión:
        # 1. Si el nodo actual es None, significa que llegamos al
final de una rama y no lo encontramos.
        # 2. Si el SKU del nodo actual es el que buscamos, lo hemos
encontrado.
        if nodo_actual is None or nodo_actual.sku == sku:
            return nodo_actual

        # Pasos recursivos:
        if sku < nodo_actual.sku:
            # Si el SKU que buscamos es menor, solo puede estar en el
sub-árbol izquierdo.
            return self._buscar_recursivo(nodo_actual.izquierda, sku)
        else:
            # Si es mayor, solo puede estar en el sub-árbol derecho.
            return self._buscar_recursivo(nodo_actual.derecha, sku)

    def eliminar_componente(self, sku):
        """Método público para eliminar un componente."""
        self.raiz = self._eliminar_recursivo(self.raiz, sku)
```

```
    def _eliminar_recursivo(self, nodo_actual, sku):
        """
        Método privado y recursivo para encontrar y eliminar un nodo.
        Este es el método más complejo debido a los diferentes casos
a manejar.
        """
        # Primero, buscamos el nodo a eliminar. Si no lo encontramos,
no hacemos nada.
        if nodo_actual is None:
            return nodo_actual

        if sku < nodo_actual.sku:
            nodo_actual.izquierda =
self._eliminar_recursivo(nodo_actual.izquierda, sku)
        elif sku > nodo_actual.sku:
            nodo_actual.derecha =
self._eliminar_recursivo(nodo_actual.derecha, sku)
        else:
            # Una vez encontrado el nodo a eliminar, analizamos los
casos.

            # Caso 1: El nodo tiene un solo hijo (o ninguno).
            # Si el hijo izquierdo es nulo, reemplazamos el nodo
actual por su hijo derecho.
            if nodo_actual.izquierda is None:
                return nodo_actual.derecha
            # Si el hijo derecho es nulo, reemplazamos el nodo actual
por su hijo izquierdo.
            elif nodo_actual.derecha is None:
                return nodo_actual.izquierda

            # Caso 2: El nodo tiene dos hijos.
            # Esta es la parte más compleja. La estrategia es:
            # 1. Encontrar el sucesor inorden (el nodo con el valor
más pequeño en el subárbol derecho).
            temp = self._encontrar_minimo(nodo_actual.derecha)
            # 2. Copiar el contenido del sucesor al nodo que queremos
```

```
"eliminar".
            nodo_actual.sku = temp.sku
            nodo_actual.nombre = temp.nombre
            nodo_actual.stock = temp.stock
            # 3. Eliminar el nodo sucesor (que es más fácil, ya que
tendrá 0 o 1 hijo).
            nodo_actual.derecha =
self._eliminar_recursivo(nodo_actual.derecha, temp.sku)

        return nodo_actual

    def _encontrar_minimo(self, nodo):
        """Función auxiliar para encontrar el nodo con el SKU más
bajo en un subárbol."""
        # Por la propiedad del BST, este siempre será el nodo más a
la izquierda.
        actual = nodo
        while actual.izquierda is not None:
            actual = actual.izquierda
        return actual

    def mostrar_inventario_ordenado(self):
        """Imprime el inventario en orden. Llama al método de
recorrido inorden."""
        print("\n--- Inventario de Componentes (Ordenado por SKU)
---")
        self._inorden_recursivo(self.raiz)
        print("-------------------------------------------------")

    def _inorden_recursivo(self, nodo_actual):
        """
        Recorre el árbol en "inorden" (Izquierda, Raíz, Derecha).
        Para un BST, este recorrido siempre devuelve los nodos en
orden ascendente.
        """
        if nodo_actual is not None:
            # 1. Ve a la izquierda recursivamente.
            self._inorden_recursivo(nodo_actual.izquierda)
```

```
            # 2. Visita (imprime) el nodo actual.
            print(nodo_actual)
            # 3. Ve a la derecha recursivamente.
            self._inorden_recursivo(nodo_actual.derecha)


#
=================================================================
=========
# BLOQUE DE EJECUCIÓN PRINCIPAL
# Este código solo se ejecuta si este archivo es el programa
principal.
#
=================================================================
=========

if __name__ == "__main__":

    # 1. Crear una instancia del árbol de inventario
    inventario = ArbolInventario()

    # 2. Agregar componentes de prueba
    print("Agregando componentes al inventario...")
    inventario.agregar_componente(2010, "Microcontrolador ESP32",
150)
    inventario.agregar_componente(1050, "Arduino Nano", 80)
    inventario.agregar_componente(3033, "Resistencia 1k ohm", 1200)
    inventario.agregar_componente(4001, "Capacitor Cerámico 100nF",
2500)
    inventario.agregar_componente(1025, "Sensor de Temperatura
DHT11", 95)
    inventario.agregar_componente(3010, "Potenciómetro 10k", 300)

    # 3. Mostrar el inventario inicial ordenado
    inventario.mostrar_inventario_ordenado()

    # 4. Probar la búsqueda de un componente
    print("\nBuscando componente con SKU 1025...")
```

```
    componente_encontrado = inventario.buscar_componente(1025)
    if componente_encontrado:
        print(f"Componente encontrado: {componente_encontrado}")
    else:
        print("Componente no encontrado.")

    # 5. Probar la eliminación de un componente
    print("\nEliminando componente con SKU 1050...")
    inventario.eliminar_componente(1050)

    # 6. Mostrar el inventario final para verificar la eliminación
    inventario.mostrar_inventario_ordenado()

    # 7. Probar a buscar un componente que no existe
    print("\nBuscando componente con SKU 9999 (inexistente)...")
    componente_no_existente = inventario.buscar_componente(9999)
    if componente_no_existente:
        print(f"Componente encontrado: {componente_no_existente}")
    else:
        print("Componente no encontrado. (Funcionamiento esperado)")
```

4. Metodología Utilizada

Para la realización de este trabajo, adoptamos una metodología colaborativa y estructurada:

1. **Investigación y Fundamentación:** Realizamos una investigación bibliográfica sobre el tema "Árboles" utilizando el material de cátedra y documentación oficial.
2. **División de Tareas:** Acordamos una distribución de responsabilidades. Ignacio Malatesta se encargó de desarrollar los conceptos teóricos fundamentales de los árboles y los algoritmos de recorrido. Pablo Limardo se enfocó en la explicación de los diferentes tipos de árboles y asumió el desarrollo, documentación y explicación del caso práctico en Python.
3. **Desarrollo y Codificación:** Se desarrolló el código en Python utilizando Visual Studio Code.
4. **Integración y Pruebas:** Integramos el código y la documentación, y realizamos pruebas conjuntas para validar el correcto funcionamiento de todas las funcionalidades.
5. **Revisión Final:** Llevamos a cabo una revisión cruzada de todo el documento para

garantizar la cohesión y calidad del informe final.

5. Resultados Obtenidos

La ejecución del caso práctico valida la correcta implementación del Árbol de Búsqueda Binaria. Al ejecutar el código con un set de datos de prueba, se demostró que:

- La función de **agregar** inserta correctamente los nodos manteniendo la propiedad de orden del BST.
- El recorrido **inorden** lista exitosamente los componentes ordenados de forma ascendente por su SKU.
- Las operaciones de **búsqueda** y **eliminación** funcionan como se espera, localizando y extrayendo nodos sin corromper la estructura del árbol.

6. Conclusión

La realización de este trabajo integrador nos permitió no solo afianzar los conocimientos teóricos sobre las estructuras de datos no lineales, sino también comprender su inmenso valor práctico en la programación. La elección del tema de árboles fué una muy buena elección, ya que nos enfrentó a un nivel de complejidad algorítmica superior al de las estructuras lineales de las primeras unidades, obligándonos a pensar en términos de recursividad, jerarquía y eficiencia.

A través del desarrollo del caso práctico, aprendimos a implementar una estructura de datos compleja desde cero en Python, a traducir algoritmos teóricos en código funcional y a valorar la importancia de elegir la estructura de datos adecuada para un problema específico.

La principal dificultad que encontramos fue la implementación del algoritmo de eliminación, particularmente la lógica para reemplazar un nodo con dos hijos por su sucesor inorden. Resolver este desafío mediante código nos sacó canas verdes, pero fue una experiencia de aprendizaje sumamente valiosa.

En definitiva, este trabajo ha sido una excelente oportunidad para aplicar la teoría en un proyecto tangible y nos proporcionó una base sólida para seguir avanzando en la tecnicatura.

7. Bibliografía

- Material de Cátedra de Programación I (2025). *Apunte teórico sobre árboles* y *Datos Avanzados*.
- Documentación de *Python* https://docs.python.org/3/

8. Anexos

- **Anexo A: Enlace al Repositorio de Código Fuente**
  - El código completo del proyecto se encuentra disponible en el siguiente repositorio de GitHub:
  - Enlace a repositorio
- **Anexo B: Enlace al Video Explicativo**
  - Se adjunta el enlace al video tutorial donde nos presentamos, explicamos el marco teórico y demostramos el funcionamiento del caso práctico:
  - Enlace al video