TRABAJO PRÁCTICO -
Implementación de árboles en Python con Listas

Programación I

Profesor: Julieta Trape

Alumnos:

- Pascutti, Valentina - valenpascutti2612@gmail.com
- Ramos, Angela Irina - ramosangelairina@gmail.com

FECHA RE-ENTREGA
20/06/2025

ÍNDICE

1. INTRODUCCIÓN

En el ámbito de la informática, las estructuras de datos constituyen pilares fundamentales para la organización y gestión eficiente de la información. Dentro de este panorama, los árboles se presentan como una de las estructuras más robustas y adaptables, cuyas aplicaciones se ramifican desde la gestión de archivos en sistemas operativos y la administración de bases de datos, hasta la optimización de algoritmos y la modelización de jerarquías en diversas áreas del conocimiento.

Estas estructuras abordan de manera eficiente el desafío de representar las relaciones jerárquicas o escenarios de la vida real donde los elementos exhiben múltiples niveles de dependencia.

En este trabajo, optamos por profundizar en los árboles como estructuras de datos avanzados, con enfoque en su implementación mediante listas anidadas. Exploraremos una forma alternativa de representar árboles sin recurrir a clases u objetos, aprovechando la inherente capacidad de las listas para contener otras listas. Nuestro objetivo es comprender cómo estas estructuras, a pesar de las similitudes con las listas enlazadas en el uso de punteros, ofrecen una organización ramificada distinta y eficiente. Consideramos que el manejo de sus representaciones es crucial para el desarrollo de algoritmos más eficientes y la optimización de recursos computacionales.

2. OBJETIVOS

- Comprender la representación de árboles binarios mediante listas anidadas en Python.
- Implementar operaciones básicas sobre árboles utilizando esta representación.
- Evaluar la eficiencia y aplicabilidad del enfoque basado en listas anidadas.
- Demostrar el uso de árboles con listas en un escenario práctico.

3. MARCO TEÓRICO

Árbol: Constituye una estructura de datos no lineal donde cada nodo puede establecer conexiones con uno o múltiples nodos. Exhibe similitudes con las listas doblemente enlazadas al emplear punteros para referenciar otros elementos. No obstante, a diferencia de estas últimas, su organización lógica no es secuencial, sino ramificada, de donde deriva su nombre.

Árbol binario: Caso particular de un árbol, en el que cada nodo puede tener como máximo, 2 hijos. Es un árbol de grado dos. El nodo izquierdo representa al hijo izquierdo y el nodo derecho al hijo derecho.

Grafo: Estructura matemática que se utiliza para representar relaciones entre objetos. Está conformado por un conjunto de nodos y un conjunto de aristas.

Propiedades:

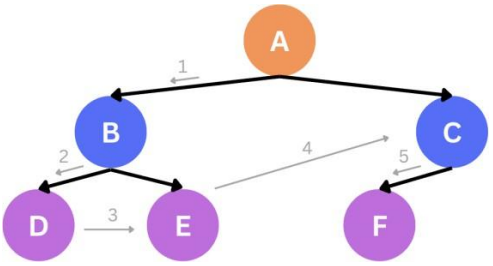
- Longitud: número de ramas que hay que transitar de un nodo a otro.
- Profundidad: longitud de camino entre el nodo raíz y el mismo
- Nivel: longitud del camino que lo conecta al nodo raíz más uno (Contiene uno o más nodos)
- Altura: es el máximo nivel del mismo.
- Grado: número de hijos que tiene dicho nodo. El grado de un árbol es el grado máximo de los nodos del árbol
- Orden: máxima cantidad de hijos que puede tener cada nodo, el orden no se calcula sino que se establece como restricción antes de construirlo
- Peso: número total de nodos que tiene un árbol, en programación saber el peso, es importante ya que podemos saber aproximadamente el tamaño del mismo y de la cantidad de memoria que ocuparía.

Formas de recorrer árboles binarios

Existen dos tipos principales de recorrido: búsqueda en profundidad (DFS) y búsqueda en amplitud (BFS).

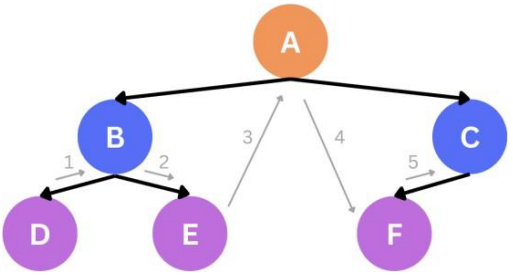
- DFS (Depth-First Search): Explora un camino hasta llegar a un nodo hoja antes de retroceder y seguir por otro camino.
- BFS (Breadth-First Search): Explora todos los nodos a un mismo nivel antes de pasar al siguiente.

- Preorden: Cuando tienes que realizar una operación en el nodo antes de procesar a sus hijos es útil utilizar el recorrido preorden. Los pasos para recorrer un árbol binario no vacío en preorden son los siguientes:



1. Se comienza por la raíz.
2. Se baja hacia el hijo izquierdo de la raíz.
3. Se recorre recursivamente el subárbol izquierdo.
4. Se sube hasta el hijo derecho de la raíz.
5. Se recorre recursivamente el subárbol derecho.

- Inorden: Es muy útil para los árboles de búsqueda binaria porque garantiza que los nodos se visiten en orden ascendente. Los pasos para recorrer un árbol binario no vacío en inorden son los siguientes:

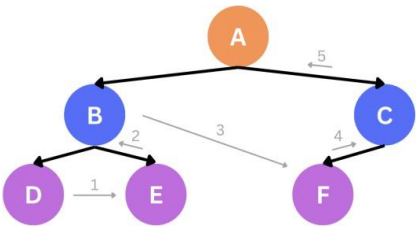


1. Se comienza por el nodo hoja que se encuentre más a la izquierda de todos.
2. Se sube hacia su nodo padre.
3. Se baja hacia el hijo derecho del nodo recorrido en el paso 2.
4. Se repiten los pasos 2 y 3 hasta terminar de recorrer el subárbol izquierdo.
5. Se visita el nodo raíz.
6. Se recorre el subárbol derecho de la misma manera que se recorre el subárbol izquierdo.

- Postorden: Este tipo de recorrido es útil para cuando quieres realizar alguna acción en los nodos hijos de un nodo antes de procesar el nodo mismo. Esta característica es particularmente relevante en el contexto de estructuras jerárquicas que exigen el procesamiento de sus componentes subordinados de forma prioritaria.

Para recorrer un árbol binario no vacío en postorden se ejecutan los siguientes pasos:

1. Se comienza por el nodo hoja que se encuentre más a la izquierda de todos.
2. Se visita su nodo hermano.
3. Se sube hacia el padre de ambos.
4. Si tuviera hermanos, visitaría su nodo hermano.
5. Se repiten los pasos 3 y 4 hasta terminar de recorrer el subárbol izquierdo.
6. Se recorre el subárbol derecho, comenzando por el nodo hoja que se encuentre más a la izquierda y siguiendo el mismo procedimiento que con el subárbol izquierdo
7. Finalmente, se visita el nodo raíz.



Insercion Y Eliminacion De Arboles Binarios

Insercion

Agregar un nuevo nodo implica encontrar una posición vacía dentro del árbol. Si el árbol está vacío, se crea el nodo raíz. Para inserciones posteriores, se recorre el árbol hasta hallar un nodo hijo (izquierdo o derecho) vacío, donde se colocará el nuevo nodo. Por convención, se prioriza insertar primero en el hijo izquierdo.

Eliminación

Eliminar un nodo consiste en retirarlo manteniendo la estructura del árbol. Primero se localiza el nodo a eliminar (usando cualquier tipo de recorrido). Luego, se reemplaza su valor por el del último nodo del árbol (generalmente la hoja más a la derecha) y se elimina ese último nodo. Es importante considerar casos especiales, como intentar eliminar de un árbol vacío.

Arboles en Python

Python permite construir y manipular árboles utilizando listas, diccionarios o DataFrames de Pandas, integrándose fácilmente con los flujos de trabajo existentes. Estas estructuras se utilizan para representar relaciones jerárquicas como árboles genealógicos, organigramas, etc. También se pueden aplicar métodos de búsqueda para localizar uno o varios nodos que cumplan con ciertos criterios, como find (para uno) o findall (para varios).

Aplicaciones de BST (Arbol de búsqueda binaria:

Algoritmos gráficos: los BST se pueden utilizar para implementar algoritmos gráficos, como en los algoritmos de árbol de expansión mínimo.
Colas de prioridad: las BST se pueden utilizar para implementar colas de prioridad, donde el elemento con mayor prioridad se encuentra en la raíz del árbol y los elementos con menor prioridad se almacenan en los subárboles.
Árbol de búsqueda binaria autoequilibrado: los BST se pueden utilizar como estructuras de datos autoequilibradas, como el árbol AVL y el árbol rojo-negro.
Almacenamiento y recuperación de datos: las BST se pueden utilizar para almacenar y recuperar datos rápidamente, como en bases de datos, donde la búsqueda de un registro específico se puede realizar en tiempo logarítmico.

Ventajas:

Búsqueda rápida: La búsqueda de un valor específico en una BST tiene una complejidad temporal promedio de O(log n), donde n es el número de nodos del árbol. Esto es mucho más rápido que buscar un elemento en un array o lista enlazada, cuya complejidad temporal es de O(n) en el peor de los casos.
Recorrido en orden: Las BST se pueden recorrer en orden, visitando el subárbol izquierdo, la raíz y el subárbol derecho. Esto permite ordenar un conjunto de datos.
Eficiente en el uso del espacio: las BST son eficientes en el uso del espacio ya que no almacenan información redundante, a diferencia de las matrices y las listas enlazadas.

Desventajas:

Árboles sesgados: si un árbol se sesga, la complejidad temporal de las operaciones de búsqueda, inserción y eliminación será O(n) en lugar de O(log n), lo que puede hacer que el árbol sea ineficiente.
Se requiere tiempo adicional: los árboles auto equilibrados requieren tiempo adicional para mantener el equilibrio durante las operaciones de inserción y eliminación.
Eficiencia: las BST no son eficientes para conjuntos de datos con muchos duplicados ya que desperdiciarán espacio.

CASO PRÁCTICO Y METODOLOGÍA UTILIZADA

El objetivo de este caso práctico es **crear un Árbol Binario de Búsqueda a partir de una lista de valores enteros ingresados por el usuario**. El programa implementa recorridos en preorden, inorden y postorden para mostrar los valores almacenados en diferentes órdenes según el tipo de recorrido. Además, incluye una función que dibuja visualmente la estructura del árbol en la consola. También valida los datos ingresados, ignorando aquellos que no sean números enteros válidos (positivos o negativos).

Para el desarrollo de este trabajo se siguieron los siguientes pasos y se utilizaron diversos recursos:

Etapas de diseño y pruebas del código, se implementaron varias funciones que permitieron construir y visualizar el árbol binario de búsqueda de manera dinámica a partir de los valores ingresados por el usuario:

- Se definió la clase **Nodo**, que representa la estructura básica del árbol. Cada nodo contiene un valor y dos referencias: una al hijo izquierdo y otra al hijo derecho.
- La función insertar permite agregar elementos al árbol respetando las reglas de un árbol binario de búsqueda. Se aplica de forma recursiva hasta encontrar la posición correcta para el nuevo valor.
- La función **postorden** realiza un recorrido en postorden del árbol, es decir, visita primero el subárbol izquierdo, luego el derecho y finalmente el nodo actual.
- La función **inorden** realiza un recorrido en inorden del árbol, es decir, visita primero el subárbol izquierdo, luego el nodo actual y finalmente el subárbol derecho.
- La función **preorden** realiza un recorrido en preorden del árbol, es decir, visita primero el nodo actual, luego el subárbol izquierdo y finalmente el derecho.
- La función **validar_entrada** verifica que los valores ingresados sean números enteros (positivos o negativos), los convierte y devuelve una lista con los válidos, ignorando los que no lo sean.
- La función **dibujar_arbol** muestra el árbol de forma visual en la consola. Utiliza caracteres especiales para representar gráficamente las conexiones entre nodos, facilitando la comprensión de su estructura jerárquica.
- La función **main** cumple el rol de controlador principal del programa. Su tarea es recibir los datos del usuario, procesarlos y coordinar el funcionamiento de las demás funciones. Solicita al usuario un conjunto de números separados por comas; una vez validados, los inserta en el árbol binario de búsqueda. Luego, se encarga de mostrar el recorrido que el usuario elija y de generar una representación visual del árbol.

Herramientas: el proyecto se desarrolló utilizando Python 3.11.9 en Visual Studio Code. No se emplearon librerías externas, solo funcionalidades estándar de Python. Mediante un repositorio Git alojado en GitHub, se gestionó el control de versiones y la colaboración, facilitando el trabajo en equipo, la revisión de código y el seguimiento de cambios.

Trabajo colaborativo: El proyecto se desarrolló de manera colaborativa entre ambas alumnas. Se utilizó Git para el control de versiones y la gestión del código fuente. El trabajo se dividió equitativamente, permitiendo optimizar tiempos y avanzar de forma organizada. Además, mantuvimos comunicación mediante videollamadas, facilitando la toma de decisiones conjunta y la resolución inmediata de posibles inconvenientes.

PROGRAMA EN PYTHON:

MODULO PRINCIPAL:

```python
from arbol import insertar, postorden, inorden, preorden
from utils import dibujar_arbol, validar_entrada

# Programa principal.
def main():
    """Función principal que solicita al usuario los valores del árbol y ejecuta las operaciones."""
    print("Bienvenido al programa de Árbol Binario de Búsqueda.")

    entrada = input("Ingresá los valores del árbol separados por comas (ej: 8,3,10,1,6): ")
    valores = validar_entrada(entrada)

    if not valores:
        print("No se ingresaron valores válidos.")
        exit()

    raiz = None
    for v in valores:
        raiz = insertar(raiz, v)

    print("\nElegí el recorrido que prefieras:")
    print("1. Postorden")
    print("2. Inorden")
    print("3. Preorden")
    opcion = input("Opción: ")

    if opcion == "1":
        postorden(raiz)
    elif opcion == "2":
        inorden(raiz)
    elif opcion == "3":
        preorden(raiz)
    else:
        print("Opción no válida.")

    print("\n\nÁrbol binario representado visualmente:")
    dibujar_arbol(raiz)

    continuar = input("\n¿Deseás crear otro árbol? (s/n): ").strip().lower()
    if continuar != 's':
        print("¡Gracias por usar el programa!")
        return

if __name__ == "__main__":
    main()
```

MODULO UTILS:

```python
def dibujar_arbol(nodo, prefijo="", es_izq=True):
    """Dibuja el árbol binario de búsqueda de forma visual (de arriba hacia abajo)."""
    if nodo is not None:
        if nodo.der:
            dibujar_arbol(nodo.der, prefijo + "    ", False)
        print(prefijo + ("└── " if es_izq else "┌── ") + str(nodo.valor))
        if nodo.izq:
            dibujar_arbol(nodo.izq, prefijo + "    ", True)


def validar_entrada(entrada):
    """Valida la entrada del usuario y convierte los valores a enteros."""
    valores = []
    for v in entrada.split(","):
        v = v.strip()
        if v.isdigit() or (v.startswith('-') and v[1:].isdigit()):
            valores.append(int(v))
        else:
            print(f"Advertencia: '{v}' no es un número válido y será ignorado.")
    return valores
```

MODULO ARBOL:

```
# Programa: Árbol Binario de Búsqueda.      Angela, 4 days ago • Mejoras en el código …
# Crear un árbol binario de búsqueda a partir de una lista de valores enteros ingresados por el usuario.
# Implementa un recorrido postorden, inorden y preorden. Utiliza una función para dibujar el árbol de forma visual.

class Nodo:
    """Clase que representa un nodo en un árbol binario de búsqueda."""
    def __init__(self, valor):
        self.valor = valor
        self.izq = None
        self.der = None

def insertar(raiz, valor):
    """Inserta un nuevo valor en el árbol binario de búsqueda."""
    """Si el árbol está vacío, crea un nuevo nodo. Si no, inserta recursivamente."""
    if raiz is None:
        return Nodo(valor)
    if valor < raiz.valor:
        raiz.izq = insertar(raiz.izq, valor)
    else:
        raiz.der = insertar(raiz.der, valor)
    return raiz

def postorden(nodo):
    """Realiza un recorrido en postorden del árbol."""
    if nodo:
        postorden(nodo.izq)
        postorden(nodo.der)
        print(nodo.valor, end=' ')

def inorden(nodo):
    """Realiza un recorrido en inorden del árbol."""
    if nodo:
        inorden(nodo.izq)
        print(nodo.valor, end=' ')
        inorden(nodo.der)

 def preorden(nodo):
     """Realiza un recorrido en preorden del árbol."""
     if nodo:
         print(nodo.valor, end=' ')
         preorden(nodo.izq)
         preorden(nodo.der)
```

RESULTADOS OBTENIDOS

El programa cumple con los objetivos planteados, brindando una herramienta educativa para comprender la construcción, recorrido y visualización de árboles binarios de búsqueda.

Se logró construir un árbol binario de búsqueda funcional que permite insertar dinámicamente valores enteros ingresados por el usuario, incluyendo números negativos, con validación y manejo de entradas inválidas mediante advertencias, garantizando que solo los datos correctos influyen en la estructura del árbol.

La implementación de los recorridos preorden, inorden y postorden se ejecutan correctamente mostrando los valores en el orden esperado según la definición de cada uno. También, la función para dibujar el árbol ofrece una representación visual clara y comprensible en la consola, facilitando la interpretación de la estructura jerárquica del árbol.

En resumen, se logró implementar satisfactoriamente un programa que permite construir y visualizar un árbol binario de búsqueda a partir de datos ingresados por el usuario.

6. CONCLUSIÓN

Para concluir, este trabajo demostró detalladamente los árboles como estructuras de datos fundamentales en la informática. Abordamos su definición como estructuras no lineales donde cada nodo puede apuntar a uno o varios, diferenciándolos de las estructuras secuenciales como las listas doblemente enlazadas por su naturaleza ramificada.

Enfocamos nuestro análisis en los árboles binarios. Esto incluye la inserción de elementos, que garantiza la integridad de la estructura de búsqueda, y los recorridos **postorden** (procesa primero los subárboles y al final la raíz), **inorden** (muestra los valores en orden ascendente) y **preorden** (recorre desde la raíz hacia las hojas).

El desarrollo del caso práctico, que posibilitó la construcción, el recorrido y la representación visual de un árbol binario de búsqueda a partir de datos suministrados por el usuario, fue clave para demostrar la aplicabilidad y la eficiencia de esta estructura. La incorporación de una función de dibujo del árbol resultó valiosa, brindando una perspectiva clara y didáctica de su organización.

En síntesis, este proyecto ha reforzado la idea de que la comprensión de los árboles es indispensable para el diseño de algoritmos eficaces y la optimización de recursos computacionales.

Bibliografía:

Julieta Trapé - Apunte teórico sobre árboles
https://www.geeksforgeeks.org/binary-search-tree-in-python/
https://builtin.com/articles/tree-python

ANEXOS

- Capturas del programa funcionando.

Se ingresan números enteros válidos para comprobar su correcto funcionamiento.

```
Ingresá los valores del árbol separados por comas (ej: 8,3,10,1,6): 6,3,9,1,4,7,11

Elegí el recorrido que prefieras:
1. Postorden
2. Inorden
3. Preorden
Opción: 1
1 4 3 7 11 9 6

Árbol binario representado visualmente:
        ┌─ 11
    ┌─ 9
        └─ 7
└─ 6
        ┌─ 4
    └─ 3
        └─ 1

¿Deseás crear otro árbol? (s/n): s
```

Se ingresó un valor no válido para comprobar las validaciones.

```
Ingresá los valores del árbol separados por comas (ej: 8,3,10,1,6): 10,5,15,2,7,12,20,uno
Advertencia: 'uno' no es un número válido y será ignorado.

Elegí el recorrido que prefieras:
1. Postorden
2. Inorden
3. Preorden
Opción: 2
2 5 7 10 12 15 20

Árbol binario representado visualmente:
        ┌─ 20
    ┌─ 15
        └─ 12
└─ 10
        ┌─ 7
    └─ 5
        └─ 2

¿Deseás crear otro árbol? (s/n): s
```

```
Ingresá los valores del árbol separados por comas (ej: 8,3,10,1,6): 8,4,12,2,6,10,14

Elegí el recorrido que prefieras:
1. Postorden
2. Inorden
3. Preorden
Opción: 3
8 4 2 6 12 10 14

Árbol binario representado visualmente:
        ┌── 14
    ┌── 12
        └── 10
└── 8
        ┌── 6
    └── 4
        └── 2

¿Deseás crear otro árbol? (s/n): n
¡Gracias por usar el programa!
```