

Trabajo Práctico Integrador – Programación I

Árboles en Python – Implementación y recorridos en un ABB

Alumnos

Axel Nicolas, Gómez Sosa – axelgo.sosa@gmail.com

Lis Araceli, Lezcano Thomas –lis.lezcano@tupad.utn.edu.ar

Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional.

Programación I

Docente Titular

Julieta Trapé

Docente Tutor

Tomás Ferro

09 de Junio de 2025

Índice

1. Introducción	3
2. Marco Teórico	4
i. Componentes básicos de un árbol	
ii. Propiedades numéricas	
iii. Tipos comunes de árboles	
iv. Aplicaciones prácticas	
v. Representación en Python	
vi. Recorridos clásicos	
3. Caso Práctico	8
i. Descripción del problema	
ii. Implementación en Python	
4. Metodología Utilizada	13
i. Investigación previa	
ii. Desarrollo del código	
iii. Herramientas utilizadas	
iv. Trabajo en equipo	
5. Resultados Obtenidos	15
6. Conclusiones	16
7. Bibliografía	17
8. Anexos	18

Trabajo Práctico Integrador

1. Introducción

El presente trabajo se centra en el estudio de los árboles como estructuras de datos fundamentales en la programación. La elección de este tema responde a su amplia aplicabilidad en diversos ámbitos de la informática, desde la representación de jerarquías hasta la implementación de algoritmos eficientes de búsqueda y ordenamiento.

Los árboles permiten modelar relaciones jerárquicas entre elementos, siendo esenciales en la construcción de sistemas de archivos, bases de datos, compiladores y en la resolución de problemas complejos mediante estructuras como los árboles de decisión. Su importancia radica en la capacidad de organizar y acceder a la información de manera estructurada y eficiente. Estos conceptos fueron abordados durante la cursada mediante clases teóricas, materiales complementarios y videos didácticos proporcionados por la cátedra.

En este trabajo se propone profundizar tanto en la teoría como en la práctica de los árboles, con especial énfasis en los árboles binarios de búsqueda (ABB), abordando sus principales propiedades, formas de recorrido e implementación en Python. Además, se explora cómo estas estructuras pueden ser representadas de distintas maneras, como mediante listas anidadas o clases, evaluando sus ventajas y limitaciones.

Este proyecto no solo busca evidenciar la comprensión de los conceptos trabajados durante la cursada, sino también reflexionar sobre el valor pedagógico de estas estructuras en la formación como programadores y su potencial aplicación en problemas reales. Estos conceptos fueron desarrollados a lo largo de la cursada mediante clases teóricas, apuntes de cátedra y videos provistos por el equipo docente.

2. Marco Teórico

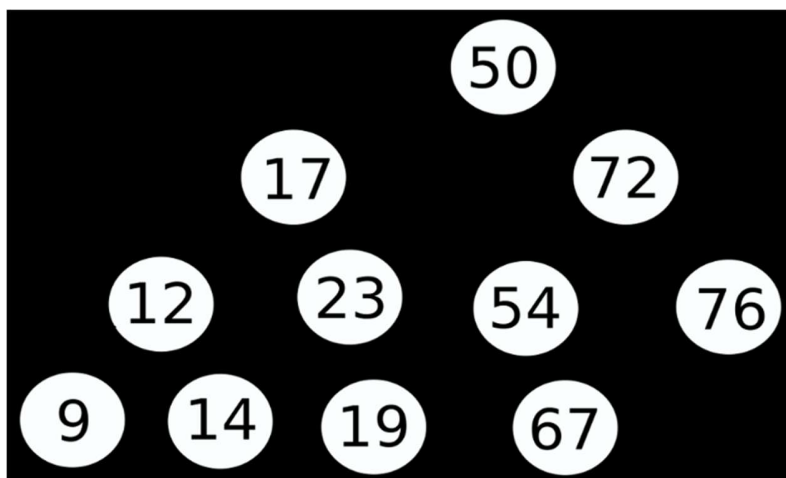
En ciencias de la computación, un árbol es una estructura de datos no lineal que permite representar relaciones jerárquicas entre elementos. Se trata de un tipo abstracto de dato (TAD) que se utiliza ampliamente en programación, bases de datos, inteligencia artificial, compiladores y muchas otras aplicaciones.

Desde una perspectiva formal, un árbol puede definirse como un grafo acíclico y conectado, donde existe un nodo especial denominado raíz, y cada nodo (excepto la raíz) tiene exactamente un nodo padre. A partir de esta raíz se desprenden subárboles que representan todos sus descendientes.

i. Componentes básicos de un árbol

- Raíz: nodo inicial y único sin padre.
- Nodo padre e hijo: relación jerárquica entre un nodo superior y su descendiente.
- Nodos hermanos: comparten el mismo nodo padre.
- Nodo interno: nodo con al menos un hijo.
- Nodo hoja: nodo sin hijos, ubicado en los extremos.
- Camino: secuencia de nodos conectados por ramas.
- Subárbol: cualquier nodo junto con sus descendientes.

Figura 1. Ejemplo de árbol binario con nodos etiquetados. Cada nodo tiene hasta dos hijos (izquierdo y derecho), y el nodo 50 es la raíz.



En esta figura, los nodos 9, 14, 19, 67 y 76 son hojas; 17, 23, 54 y 72 son nodos internos; y 12 y 23 son hermanos.

ii. Propiedades numéricas

- Nivel: cantidad de pasos desde la raíz. Por convención, la raíz está en el nivel 1.
- Profundidad: cantidad de ramas desde la raíz a un nodo (nivel - 1).
- Altura del árbol: nivel máximo alcanzado.
- Grado de un nodo: número de hijos que tiene.
- Grado del árbol: grado máximo entre todos los nodos.
- Orden del árbol: número máximo de hijos permitidos por nodo (en un árbol binario, es 2).
- Peso: cantidad total de nodos del árbol.

Una analogía útil es imaginar que el árbol es un edificio: la raíz está en planta baja, y los niveles aumentan a medida que se asciende por la jerarquía.

iii. Tipos comunes de árboles

- Árbol binario: cada nodo tiene como máximo dos hijos.

- Árbol binario de búsqueda (ABB o BST): estructura donde los valores del subárbol izquierdo son menores que el nodo, y los del derecho, mayores.
- Árboles auto-balanceados (AVL, rojo-negro): estructuras que controlan su altura para mantener eficiencia en búsquedas, inserciones y eliminaciones ($O(\log n)$).
- Árbol N-ario o K-ario: cada nodo puede tener hasta K hijos. Usado en estructuras como árboles B o B+ (índices de bases de datos).
- Montículo (heap): árbol binario casi completo usado en algoritmos como heap sort.
- Trie: árbol especializado para búsquedas de prefijos o secuencias, útil en autocompletado.

iv. Aplicaciones prácticas

Los árboles son esenciales para múltiples usos en programación:

- Sistemas de archivos: jerarquías de carpetas.
- Bases de datos: uso de árboles B/B+ para optimizar el acceso a discos.
- Inteligencia Artificial: árboles de decisión y árboles de juego.
- Compresión de datos: árboles de Huffman.
- Expresiones matemáticas: árboles de operadores.
- Procesamiento de texto: tries para búsqueda de palabras.

v. Representación en Python

Aunque Python no posee una estructura de árbol incorporada, existen dos formas comunes de representarlos:

1- Mediante clases

```
class Nodo:
    def __init__(self, valor):
        self.valor = valor
        self.izquierda = None
        self.derecha = None
```

2- Mediante listas anidadas:

```
['A', ['B', ['D', [], []], ['E', [], []]], ['C', ['F', [], []], ['G',
[], []]]]
```

Esta última forma, vista en los videos del curso, permite trabajar sin clases, ideal para una introducción más didáctica.

vi. Recorridos clásicos

Para visitar los nodos de un árbol, existen tres formas principales:

- Inorden (izquierda - raíz - derecha): muestra los elementos ordenados (en un ABB).
- Preorden (raíz - izquierda - derecha): útil para copiar estructuras.
- Postorden (izquierda - derecha - raíz): útil para eliminar estructuras o evaluar expresiones.

Ejemplo con recorrido inorden del árbol de la Figura 1:
9, 12, 14, 17, 19, 23, 50, 54, 67, 72, 76.

Este marco teórico fue abordado a través de contenidos del módulo “Datos Avanzados”, material oficial y ejemplos prácticos trabajados en clase.

3. Caso Practico

i. Descripción del problema

El objetivo de este caso práctico es implementar un Árbol Binario de Búsqueda (ABB) en Python, permitiendo realizar operaciones fundamentales como la inserción de valores, búsqueda de elementos y recorridos en diferentes órdenes (inorden, preorden y postorden).

Se busca simular el funcionamiento de un ABB en un contexto realista, donde los datos ingresados se almacenan de forma jerárquica y ordenada. Esto permite optimizar tareas como búsquedas o recorridos estructurados, fundamentales en programación y estructuras de datos.

ii. Implementación en Python

A continuación, se presenta la implementación completa de un Árbol Binario de Búsqueda (ABB) en Python. El programa permite insertar elementos, recorrer la estructura en diferentes órdenes y realizar búsquedas para validar su funcionamiento.

1- Definición de la clase Nodo y funciones principales

```
# Clase que define un nodo del árbol
class Nodo:
    def __init__(self, valor):
        self.valor = valor
        self.izquierda = None # Subárbol izquierdo
        self.derecha = None   # Subárbol derecho

# Función para insertar un valor en el ABB
def insertar(nodo, valor):
    if nodo is None:
        return Nodo(valor)
    if valor < nodo.valor:
        nodo.izquierda = insertar(nodo.izquierda, valor)
    else:
        nodo.derecha = insertar(nodo.derecha, valor)
    return nodo
```



```
# Función para buscar un valor dentro del ABB
def buscar(nodo, valor):
    if nodo is None:
        return None
    if valor == nodo.valor:
        return nodo
    elif valor < nodo.valor:
        return buscar(nodo.izquierda, valor)
    else:
        return buscar(nodo.derecha, valor)
```

2- Recorridos del árbol

```
def inorden(nodo):
    if nodo:
        inorden(nodo.izquierda)
        print(nodo.valor, end=' ')
        inorden(nodo.derecha)

def preorden(nodo):
    if nodo:
        print(nodo.valor, end=' ')
        preorden(nodo.izquierda)
        preorden(nodo.derecha)

def postorden(nodo):
    if nodo:
        postorden(nodo.izquierda)
        postorden(nodo.derecha)
        print(nodo.valor, end=' ')
```

3- Creación del árbol y prueba de funcionalidades

```
valores = [50, 30, 70, 20, 40, 60, 80]
raiz = None

# Inserción de valores
for valor in valores:
    raiz = insertar(raiz, valor)

# Recorridos
print("Recorrido inorden:")
inorden(raiz) # Esperado: 20 30 40 50 60 70 80

print("\nRecorrido preorden:")
preorden(raiz) # Esperado: 50 30 20 40 70 60 80
```

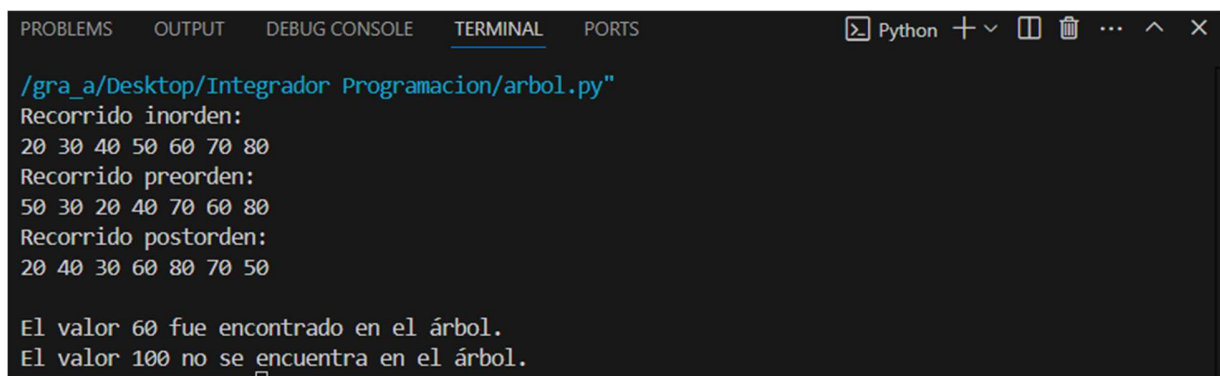
```
print("\nRecorrido postorden:")
postorden(raiz) # Esperado: 20 40 30 60 80 70 50

# Búsqueda
valor_buscado = 60
resultado = buscar(raiz, valor_buscado)

if resultado:
    print(f"\n\nEl valor {valor_buscado} fue encontrado en el árbol.")
else:
    print(f"\n\nEl valor {valor_buscado} no se encuentra en el árbol.")
```

4- Capturas

La siguiente serie de capturas muestra la ejecución del código y permite validar su funcionamiento a través de la salida por consola.



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python + v [icon] [icon] ... ^ x

/gra_a/Desktop/Integrador Programacion/arbol.py"
Recorrido inorden:
20 30 40 50 60 70 80
Recorrido preorden:
50 30 20 40 70 60 80
Recorrido postorden:
20 40 30 60 80 70 50

El valor 60 fue encontrado en el árbol.
El valor 100 no se encuentra en el árbol.
```

Figura 2. Ejecución del Árbol Binario de Búsqueda (ABB) en Python desde VS Code. Se observa el recorrido inorden, preorden y postorden, junto con la búsqueda de un valor existente (60) y uno inexistente (100).

```
50 valores = [50, 30, 70, 20, 40, 60, 80]
51 raiz = None
52
53 # Insertar los valores
54 for valor in valores:
55     raiz = insertar(raiz, valor)
56
57 # Recorridos
58 print("Recorrido inorden:")
59 inorden(raiz)
60
61 print("\nRecorrido preorden:")
62 preorden(raiz)
63
64 print("\nRecorrido postorden:")
65 postorden(raiz)
66
67 # Búsqueda de un valor
68 valor_buscado = 60
69 resultado = buscar(raiz, valor_buscado)
70
71 if resultado:
72     print(f"\nEl valor {valor_buscado} fue encontrado en el árbol.")
73 else:
74     print(f"\nEl valor {valor_buscado} no se encuentra en el árbol.")
75
76 # Búsqueda negativa (valor que no existe)
77 valor_inexistente = 100
78 resultado = buscar(raiz, valor_inexistente)
79
80 if resultado:
81     print(f"El valor {valor_inexistente} fue encontrado en el árbol.")
82 else:
83     print(f"El valor {valor_inexistente} no se encuentra en el árbol.")
84
```

Figura 3. Bloque principal del programa en Python que construye el árbol, ejecuta los recorridos e implementa las búsquedas.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\gra_a\Desktop\Integrador Programacion> &
Recorrido inorden:
20 30 40 50 60 70 80
Recorrido preorden:
50 30 20 40 70 60 80
Recorrido postorden:
20 40 30 60 80 70 50

El valor 999 no se encuentra en el árbol.
PS C:\Users\gra_a\Desktop\Integrador Programacion>
```

Figura 4. Respuesta del Árbol Binario de Búsqueda ante la búsqueda de un valor inexistente (999), confirmando el correcto manejo de condiciones negativas.

5- Decisiones de diseño

Se optó por una implementación simple con clases para mejorar la legibilidad del código y reflejar la estructura jerárquica de un ABB. Las funciones fueron diseñadas de forma recursiva para ajustarse naturalmente al comportamiento del árbol. Los datos se cargaron desde una lista predefinida, simulando entradas de usuario.

No se utilizó un árbol auto balanceado para priorizar la comprensión conceptual antes que la optimización avanzada.

6- Validación del funcionamiento

El correcto funcionamiento del árbol binario de búsqueda se validó mediante pruebas de recorridos (inorden, preorden y postorden), observando que el orden de los nodos respetara la estructura del ABB. Además, se realizaron búsquedas de valores existentes (como el 60) y valores inexistentes (como el 100 y el 999), verificando que el programa respondiera de forma adecuada en ambos casos.

Las salidas esperadas fueron observadas en consola, y se documentaron mediante capturas de pantalla incluidas en esta sección. No se detectaron errores durante la ejecución, y cada recorrido y búsqueda arrojó los resultados esperados, lo que confirma la validez de la implementación.

4. Metodología Utilizada

Para el desarrollo del presente trabajo integrador se siguió una metodología basada en investigación teórica, desarrollo incremental del código, validación práctica y trabajo colaborativo. Además, se priorizó el uso de recursividad por su correspondencia natural con la estructura jerárquica de los árboles, facilitando una implementación clara y didáctica.

i. Investigación previa

Se consultaron materiales provistos por la cátedra, como apuntes sobre árboles binarios, representaciones con listas anidadas, y videos explicativos sobre estructuras de datos en Python. También se exploraron fuentes externas como documentación oficial de Python, ejemplos educativos y tutoriales de implementación de árboles.

ii. Desarrollo del código

Se comenzó por la definición de una estructura de nodo mediante clases, utilizando referencias a hijos izquierdo y derecho. Luego se desarrollaron funciones recursivas para insertar valores, recorrer el árbol (inorden, preorden y postorden) y buscar elementos. El programa se diseñó de forma progresiva, probando cada funcionalidad por separado y refinando el código en función de los resultados obtenidos.

iii. Herramientas utilizadas

- Lenguaje: Python 3.11
- Entorno de desarrollo: Visual Studio Code
- Capturas y validación: Terminal integrada de VS Code y herramientas nativas de Windows
- Control de versiones y colaboración: Git

- Asistencia externa: Uso de ChatGPT para depurar errores, validar sintaxis y acompañar el desarrollo del trabajo.

iv. Trabajo en equipo

El trabajo se realizó en pareja. Axel se encargó de la investigación teórica, redacción del marco teórico, desarrollo de funciones de inserción y búsqueda. Lis se encargó de implementación de recorridos, pruebas de validación, redacción de los puntos prácticos y conclusiones. La coordinación se llevó a cabo mediante reuniones virtuales y uso compartido de archivos. La colaboración permitió combinar conocimientos y asegurar una mejor comprensión de los temas trabajados.

5. Resultados Obtenidos

A lo largo del desarrollo del trabajo integrador, se obtuvo una implementación funcional de un Árbol Binario de Búsqueda (ABB) utilizando el lenguaje Python. Los recorridos inorden, preorden y postorden fueron ejecutados exitosamente, reflejando los resultados esperados según lo explicado en los apuntes de la cátedra y los videos de referencia.

En particular, el recorrido inorden devolvió los valores en orden creciente, validando la estructura jerárquica correcta del árbol. Por su parte, los recorridos preorden y postorden también presentaron el orden lógico correspondiente a sus definiciones teóricas, permitiendo visualizar cómo se navega el árbol de distintas formas.

Se realizaron búsquedas de valores previamente insertados, como el 60, y valores no existentes, como el 100 y el 999. En todos los casos, las respuestas del sistema fueron correctas: el programa identificó correctamente la existencia o ausencia de los valores en el árbol, confirmando el funcionamiento esperado del algoritmo de búsqueda.

Las salidas generadas se visualizaron en consola y fueron documentadas mediante capturas de pantalla, que se incluyen en la sección de caso práctico. Estas evidencias demuestran el éxito de la implementación tanto a nivel lógico como operativo.

Esto demuestra que la implementación cumple tanto con los objetivos teóricos planteados al inicio como con la funcionalidad esperada desde una perspectiva práctica.

6. Conclusiones

El desarrollo de este trabajo integrador permitió consolidar los conocimientos teóricos y prácticos adquiridos a lo largo de la cursada, particularmente en lo referente al uso de estructuras de datos como los árboles binarios de búsqueda.

A través de la implementación en Python, se comprendió en profundidad el funcionamiento de los recorridos clásicos (inorden, preorden y postorden) y su relación con la estructura jerárquica del árbol. La utilización de clases permitió modelar de forma clara la organización de nodos y sus relaciones, mientras que las funciones recursivas facilitaron una solución elegante y alineada con la naturaleza del problema.

Además, la validación del funcionamiento mediante pruebas concretas reforzó la importancia de diseñar algoritmos robustos y comprobables. Se evidenció que una correcta planificación, acompañada de documentación, testing y trabajo en equipo, son pilares fundamentales en el desarrollo de soluciones en programación.

Este proyecto también brindó la oportunidad de reflexionar sobre la utilidad de los árboles en escenarios reales, tales como motores de búsqueda, sistemas de archivos, jerarquías organizativas o estructuras de decisión, reafirmando su valor formativo en la carrera.

Finalmente, el trabajo en pareja permitió ejercitar la comunicación, la división de responsabilidades y el trabajo colaborativo, aspectos fundamentales en el desarrollo de software.

7. Bibliografía

Universidad Tecnológica Nacional. (2025). *Apuntes de la unidad "Árboles con listas"*. Tecnicatura Universitaria en Programación.

Universidad Tecnológica Nacional. (2025, mayo 27). *Programación – Datos Avanzados – Árboles* (1° ed.). Universidad Tecnológica Nacional.

Universidad Tecnológica Nacional. (2025). *Listas con árboles* [Video]. YouTube. <https://www.youtube.com/watch?v=-D4SxeHQGIg>

Universidad Tecnológica Nacional. (2025). *Datos Avanzados – Playlist sobre estructuras de árboles* [Lista de reproducción]. YouTube. https://www.youtube.com/playlist?list=PLy5wpwhsM-2IIY-qe_fALJ4K_XAhLZ2I-

Python Software Foundation. (n.d.). *Python 3 documentation*. <https://docs.python.org/3/>

Llamas, L. (s.f.). *¿Qué son y cómo usar los árboles?* Luis Llamas. <https://www.luisllamas.es/que-es-un-arbol>

Wikipedia. (s.f.). *Árbol (informática)*. Wikipedia. https://es.wikipedia.org/wiki/%C3%81rbol_%28inform%C3%A1tica%29

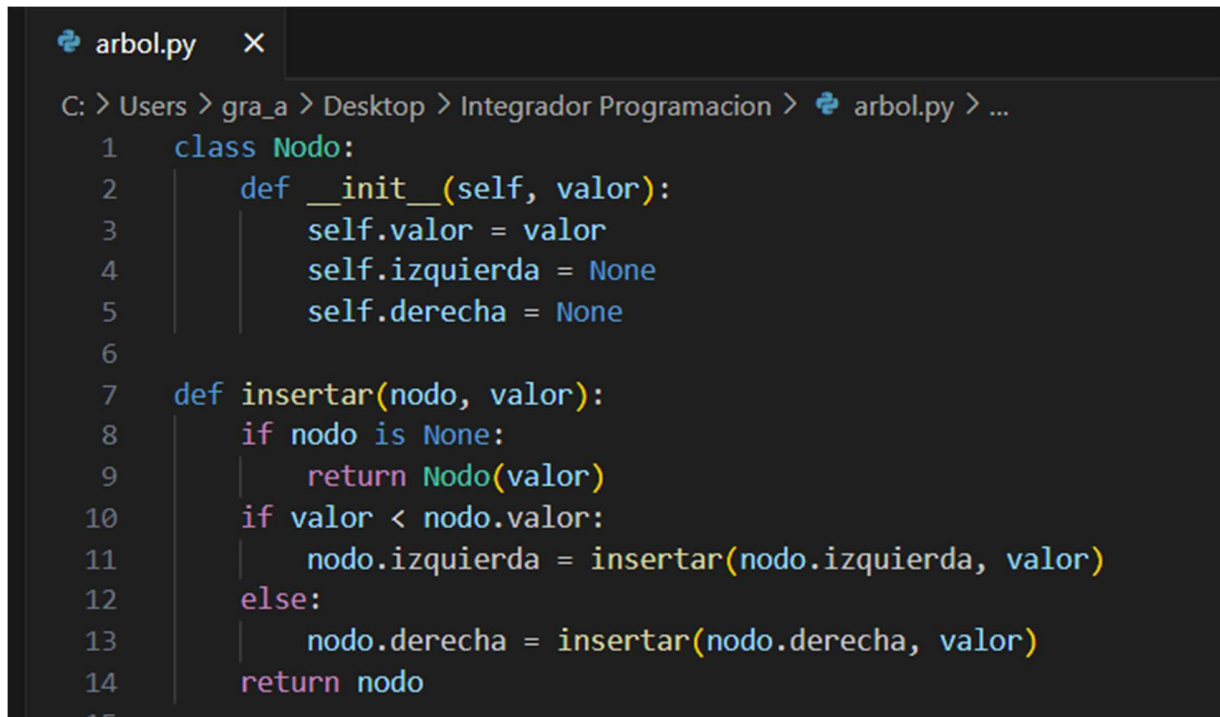
Datos Ninja. (s.f.). *Árboles de decisión en Python*. <https://datos.ninja/tutorial/arboles-decision-python/>

Tav TechSolutions. (s.f.). *Árbol binario*. <https://tavtechsolutions.com/es/glosario/%C3%A1rbol-binario>

Ciencia de Datos. (s.f.). *Árboles de decisión en Python*. https://cienciadedatos.net/documentos/py07_arboles_decision_python

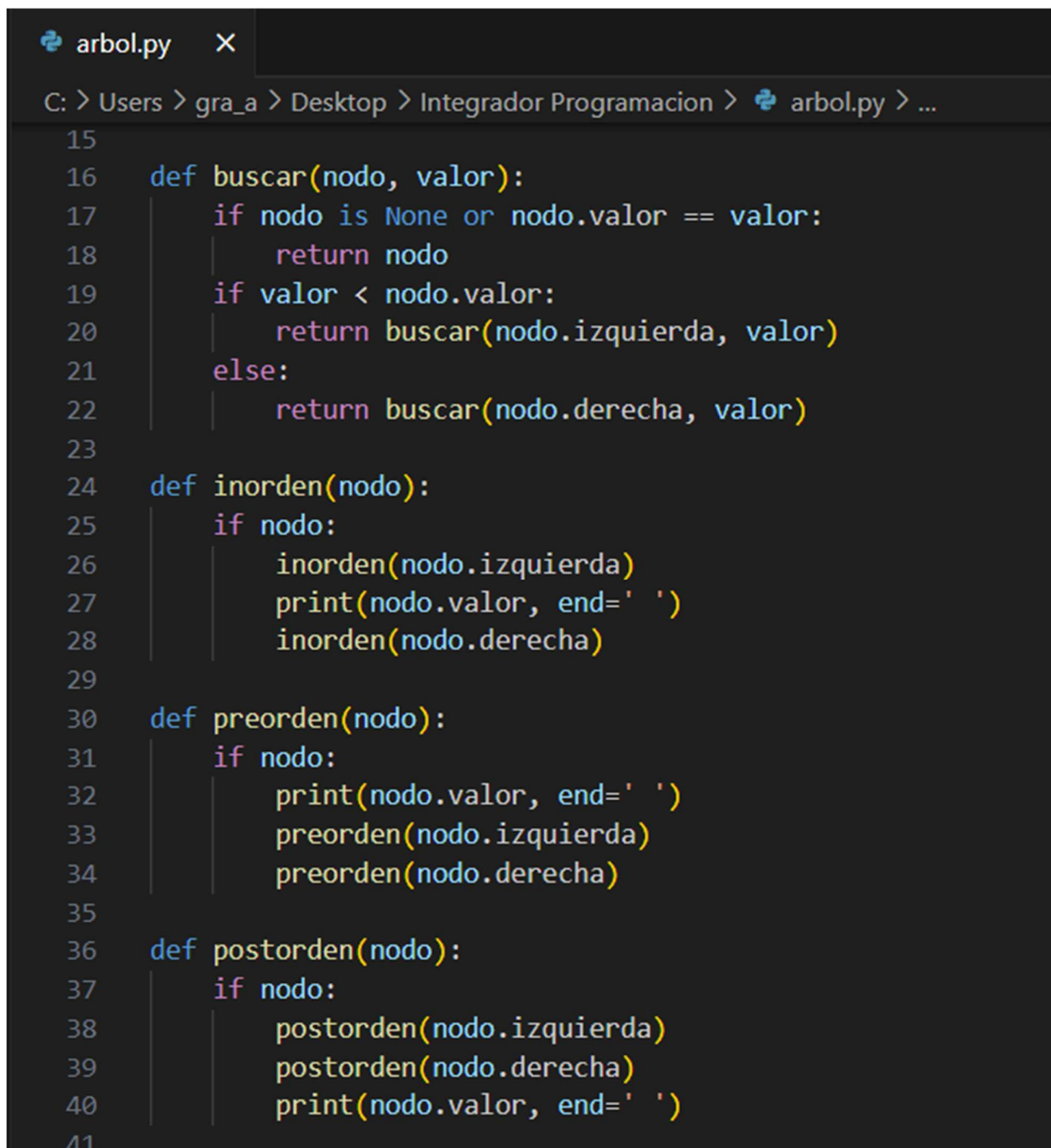
8. Anexos

A continuación, se incluyen capturas adicionales que documentan el código completo desarrollado, así como su ejecución y validación funcional. Estas imágenes complementan el Caso Práctico y refuerzan los resultados obtenidos.



```
arbol.py X
C: > Users > gra_a > Desktop > Integrador Programacion > arbol.py > ...
1 class Nodo:
2     def __init__(self, valor):
3         self.valor = valor
4         self.izquierda = None
5         self.derecha = None
6
7 def insertar(nodo, valor):
8     if nodo is None:
9         return Nodo(valor)
10    if valor < nodo.valor:
11        nodo.izquierda = insertar(nodo.izquierda, valor)
12    else:
13        nodo.derecha = insertar(nodo.derecha, valor)
14    return nodo
15
```

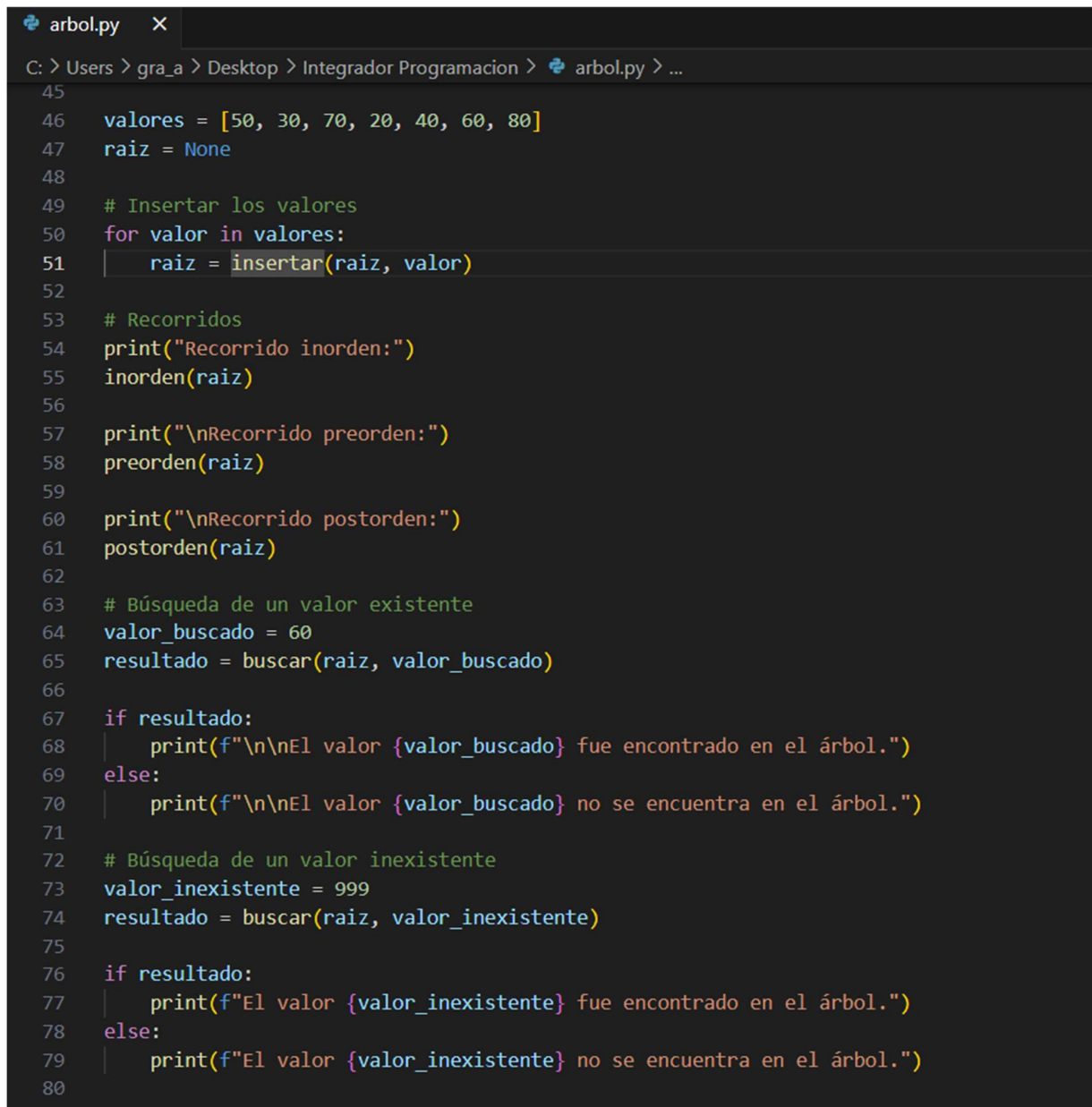
Figura 3a. Definición de la clase `Nodo` y de la función `insertar()`, que constituyen la base de la estructura del árbol binario de búsqueda (ABB) implementado en Python.



```
arbol.py X
C: > Users > gra_a > Desktop > Integrador Programacion > arbol.py > ...

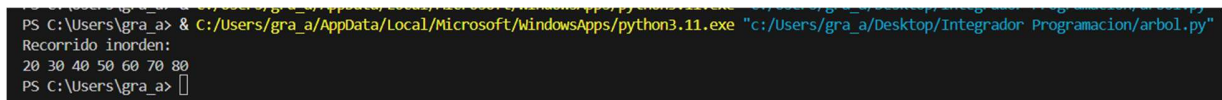
15
16 def buscar(nodo, valor):
17     if nodo is None or nodo.valor == valor:
18         return nodo
19     if valor < nodo.valor:
20         return buscar(nodo.izquierda, valor)
21     else:
22         return buscar(nodo.derecha, valor)
23
24 def inorden(nodo):
25     if nodo:
26         inorden(nodo.izquierda)
27         print(nodo.valor, end=' ')
28         inorden(nodo.derecha)
29
30 def preorden(nodo):
31     if nodo:
32         print(nodo.valor, end=' ')
33         preorden(nodo.izquierda)
34         preorden(nodo.derecha)
35
36 def postorden(nodo):
37     if nodo:
38         postorden(nodo.izquierda)
39         postorden(nodo.derecha)
40         print(nodo.valor, end=' ')
41
```

Figura 3b. Implementación de la función buscar() y los recorridos clásicos (inorden, preorden y postorden) utilizados para recorrer y validar la estructura del árbol binario.



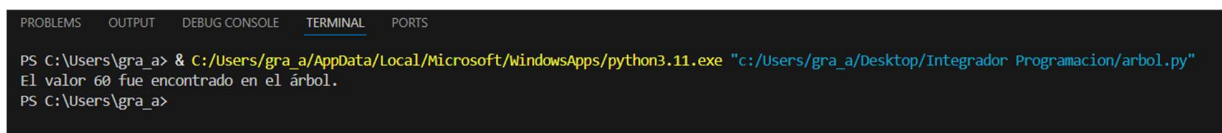
```
45
46 valores = [50, 30, 70, 20, 40, 60, 80]
47 raiz = None
48
49 # Insertar los valores
50 for valor in valores:
51     raiz = insertar(raiz, valor)
52
53 # Recorridos
54 print("Recorrido inorden:")
55 inorden(raiz)
56
57 print("\nRecorrido preorden:")
58 preorden(raiz)
59
60 print("\nRecorrido postorden:")
61 postorden(raiz)
62
63 # Búsqueda de un valor existente
64 valor_buscado = 60
65 resultado = buscar(raiz, valor_buscado)
66
67 if resultado:
68     print(f"\n\nEl valor {valor_buscado} fue encontrado en el árbol.")
69 else:
70     print(f"\n\nEl valor {valor_buscado} no se encuentra en el árbol.")
71
72 # Búsqueda de un valor inexistente
73 valor_inexistente = 999
74 resultado = buscar(raiz, valor_inexistente)
75
76 if resultado:
77     print(f"El valor {valor_inexistente} fue encontrado en el árbol.")
78 else:
79     print(f"El valor {valor_inexistente} no se encuentra en el árbol.")
80
```

Figura 3c. Sección principal del programa, donde se construye el árbol, se ejecutan los recorridos y se realizan búsquedas de valores existentes e inexistentes.



```
PS C:\Users\gra_a> & C:/Users/gra_a/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/gra_a/Desktop/Integrador Programacion/arbol.py"
Recorrido inorden:
20 30 40 50 60 70 80
PS C:\Users\gra_a>
```

Figura 5. Resultado del recorrido inorden. Los valores se muestran en orden creciente, validando la estructura y funcionamiento del Árbol Binario de Búsqueda (ABB).



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\gra_a> & C:/Users/gra_a/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/gra_a/Desktop/Integrador Programacion/arbol.py"
El valor 60 fue encontrado en el árbol.
PS C:\Users\gra_a>
```

Figura 6. Resultado de búsqueda positiva en el Árbol Binario de Búsqueda. Se confirma que el valor 60 fue correctamente localizado dentro de la estructura.