Título del trabajo: **“Arbones Binarios en Python con Listas”**

**Farid Salomón, faridsalomon90@gmail.com - Gabriel Gonzalez, gaby250995@gmail.com**

- Materia: **Programacion I**
- Profesor/a: Cinthia Rogoni, Bruselario Sebastian
- Fecha de Entrega: **20/06/2025**

# Índice

# 1. Introducción

El árbol binario es una estructura de datos jerárquica en la que cada nodo puede tener como máximo dos hijos (izquierdo y derecho). Se aplica en sistemas de archivos, algoritmos de ordenamiento y búsqueda eficiente. En este trabajo se empleará una representación con listas anidadas en Python para facilitar la comprensión de la mecánica de inserción, los recorridos y la visualización.

# 2. Marco Teórico

### Estructura de los Nodos

En un árbol binario, cada nodo contiene:

- **Un valor o dato**
- **Un enlace al subárbol izquierdo**
- **Un enlace al subárbol derecho**

En términos computacionales, esto puede representarse como una lista en Python:

```
nodo = [valor, subárbol_izquierdo, subárbol_derecho]
```

Si un nodo no tiene hijo izquierdo o derecho, se representa con `None`. Esta estructura facilita la recursividad y es muy usada en la enseñanza de estructuras de datos.

### Clasificación de los Árboles Binarios

Los árboles binarios pueden clasificarse según sus características estructurales:

- **Árbol binario completo**: todos los niveles están completamente llenos, excepto posiblemente el último, que se llena de izquierda a derecha.
- **Árbol binario lleno**: cada nodo tiene 0 o 2 hijos, nunca 1.
- **Árbol binario perfecto**: es completo y lleno, y todos los nodos hoja están en el mismo nivel.
- **Árbol binario balanceado**: la diferencia de altura entre subárboles izquierdo y derecho en cualquier nodo es a lo sumo 1.

Estas variantes afectan directamente la eficiencia de los algoritmos de búsqueda e inserción.

### Tipos de Recorridos (Traversals)

Los recorridos son algoritmos para visitar todos los nodos en un orden específico:

- **Inorden (izq → raíz → der)**
  Ideal para árboles de búsqueda binarios ya que devuelve los valores en orden creciente.
- **Preorden (raíz → izq → der)**
  Útil para copiar árboles o imprimir expresiones en notación prefija.
- **Postorden (izq → der → raíz)**
  Se usa comúnmente para eliminar o evaluar árboles de expresiones.

Estos recorridos pueden implementarse de forma **recursiva o iterativa**, según la complejidad del árbol y el lenguaje de programación utilizado.

### Complejidad Computacional

El rendimiento de un árbol binario depende de su balance:

| Operación | Árbol No Balanceado | Árbol Balanceado (AVL, Red-Black) |
|---|---|---|
| Búsqueda | O(n) | O(log n) |
| Inserción | O(n) | O(log n) |
| Eliminación | O(n) | O(log n) |

En el **peor caso**, un árbol no balanceado se comporta como una lista enlazada, perdiendo eficiencia.

### Aplicaciones de los Árboles Binarios

Los árboles binarios son fundamentales en muchas áreas de la informática:

- **Árboles binarios de búsqueda (BST)**: permiten búsquedas rápidas y eficientes.
- **Árboles de expresión**: utilizados en compiladores para representar expresiones matemáticas.
- **Árboles de decisión**: aplicados en inteligencia artificial y clasificación.
- **Estructuras jerárquicas**: como sistemas de archivos o representación de HTML/DOM.
- **Algoritmos de compresión**: como Huffman, que usa árboles binarios para codificación óptima.

### Fuentes Consultadas


- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
- Goodrich, M. T., Tamassia, R. (2014). *Data Structures and Algorithms in Python*.
- Python Docs. https://docs.python.org/3/
- GeeksForGeeks. https://www.geeksforgeeks.org/binary-tree-data-structure/
- Visualgo. https://visualgo.net


## 3. Metodología Utilizada

1. Estudio de características de árboles binarios.
2. Implementación manual de funciones sobre listas.
3. Programación recursiva de recorridos (inorden, preorden, postorden).
4. Testeo con valores predefinidos.

5. Extensión con búsqueda, cálculo de altura y recorrido por niveles.

## 4. Caso Practico: Script Inicial

```
#modulo árbol_lista.py
"""
Módulo principal: define la estructura básica del árbol y sus operaciones.
"""

def crear_arbol(valor):
    """
    Crea un nuevo árbol con el valor dado y subárboles vacíos.
    :param valor: dato a almacenar en la raíz del árbol
    :return: lista [valor, None, None]
    """
    return [valor, None, None]

def insertar_izquierda(arbol, valor):
    """
    Inserta un nuevo nodo con 'valor' como hijo izquierdo.
    Si ya había, ese subárbol pasa a ser hijo izquierdo del nuevo nodo.
    """
    if arbol[1] is None:
        arbol[1] = crear_arbol(valor)
    else:
        nuevo = crear_arbol(valor)
        nuevo[1] = arbol[1]
        arbol[1] = nuevo
```

```
def insertar_derecha(arbol, valor):
    """
    Inserta un nuevo nodo con 'valor' como hijo derecho.
    Si ya había, ese subárbol pasa a ser hijo derecho del nuevo nodo.
    """
    if arbol[2] is None:
        arbol[2] = crear_arbol(valor)
    else:
        nuevo = crear_arbol(valor)
        nuevo[2] = arbol[2]
        arbol[2] = nuevo


def imprimir_arbol(arbol, nivel=0):
    """
    Imprime el árbol rotado 90° en sentido horario.
    :param nivel: para controlar sangrías según profundidad.
    """
    if arbol is None:
        return
    imprimir_arbol(arbol[2], nivel+1)
    print('    ' * nivel + str(arbol[0]))
    imprimir_arbol(arbol[1], nivel+1)



# utilidades_arbol.py
"""
Módulo de utilidades: recorridos, búsqueda y análisis.
"""
```

```
def inorden(arbol, resultado=None):
    """Recorrido inorden: izq → raíz → der."""
    if resultado is None:
        resultado = []
    if arbol is not None:
        inorden(arbol[1], resultado)
        resultado.append(arbol[0])
        inorden(arbol[2], resultado)
    return resultado

def preorden(arbol, resultado=None):
    """Recorrido preorden: raíz → izq → der."""
    if resultado is None:
        resultado = []
    if arbol is not None:
        resultado.append(arbol[0])
        preorden(arbol[1], resultado)
        preorden(arbol[2], resultado)
    return resultado

def postorden(arbol, resultado=None):
    """Recorrido postorden: izq → der → raíz."""
    if resultado is None:
        resultado = []
    if arbol is not None:
        postorden(arbol[1], resultado)
        postorden(arbol[2], resultado)
        resultado.append(arbol[0])
    return resultado
```

```
def buscar(arbol, valor):
    """Retorna True si 'valor' existe en el árbol."""
    if arbol is None:
        return False
    if arbol[0] == valor:
        return True
    return buscar(arbol[1], valor) or buscar(arbol[2], valor)


def altura(arbol):
    """Retorna la altura máxima del árbol."""
    if arbol is None:
        return 0
    return 1 + max(altura(arbol[1]), altura(arbol[2]))


def nivel_por_nivel(arbol):
    """Recorrido por niveles (BFS)."""
    if arbol is None:
        return []
    cola, resultado = [arbol], []
    while cola:
        nodo = cola.pop(0)
        resultado.append(nodo[0])
        if nodo[1]:
            cola.append(nodo[1])
        if nodo[2]:
            cola.append(nodo[2])
    return resultado.
```

```
# main.py
from arbol_listas import crear_arbol, insertar_izquierda, insertar_derecha, imprimir_arbol
from utilidades_arbol import inorden, preorden, postorden, buscar, altura, nivel_por_nivel

def menu():
    print("\nOpciones:")
    print("1. Insertar nodo")
    print("2. Imprimir árbol")
    print("3. Recorridos (inorden, preorden, postorden)")
    print("4. Buscar un valor")
    print("5. Altura del árbol")
    print("6. Recorrido por niveles")
    print("0. Salir")

def buscar_nodo(arbol, valor):
    """Busca el nodo con ese valor y lo retorna (referencia)."""
    if arbol is None:
        return None
    if arbol[0] == valor:
        return arbol
    izquierdo = buscar_nodo(arbol[1], valor)
    if izquierdo:
        return izquierdo
    return buscar_nodo(arbol[2], valor)

if __name__ == "__main__":
    raiz = None
```

```
while True:
    menu()
    opcion = input("Seleccione una opción: ")

    if opcion == "1":
        valor = int(input("Ingrese el valor a insertar: "))
        if raiz is None:
            raiz = crear_arbol(valor)
            print("Raíz creada.")
        else:
            padre = int(input("Ingrese el valor del nodo padre: "))
            direccion = input("¿Insertar a la izquierda (i) o derecha (d)?: ").lower()
            nodo_padre = buscar_nodo(raiz, padre)
            if nodo_padre:
                if direccion == "i":
                    insertar_izquierda(nodo_padre, valor)
                    print(f"{valor} insertado a la izquierda de {padre}.")
                elif direccion == "d":
                    insertar_derecha(nodo_padre, valor)
                    print(f"{valor} insertado a la derecha de {padre}.")
                else:
                    print("Dirección inválida.")
            else:
                print("Nodo padre no encontrado.")

    elif opcion == "2":
        if raiz:
            imprimir_arbol(raiz)
        else:
```

```
        print("El árbol está vacío.")

elif opcion == "3":
    if raiz:
        print("Inorden:", inorden(raiz))
        print("Preorden:", preorden(raiz))
        print("Postorden:", postorden(raiz))
    else:
        print("El árbol está vacío.")

elif opcion == "4":
    valor = int(input("Valor a buscar: "))
    encontrado = buscar(raiz, valor)
    print(f"Resultado: {'Sí' if encontrado else 'No'}")

elif opcion == "5":
    print("Altura del árbol:", altura(raiz))

elif opcion == "6":
    print("Recorrido por niveles:", nivel_por_nivel(raiz))

elif opcion == "0":
    print("Programa finalizado.")
    break
else:
    print("Opción inválida. Intente nuevamente.")
```

## 5. Resultados Obtenidos

- El árbol binario se construyó correctamente con nodos insertados según su valor.
- Los recorridos Inorden, Preorden y Postorden se realizaron de manera correcta.
- Búsqueda y cálculo de altura funcionaron adecuadamente.
- La visualización con `imprimir_arbol` muestra la estructura jerárquica rotada 90°.

## 6. Conclusiones

El uso de árboles binarios en Python facilita la organización de datos de manera eficiente. Los algoritmos de recorrido permiten explorar la estructura según las necesidades de cada problema.

**Reflexión:** Aprender estructuras de datos como árboles mejora las habilidades de programación y fomenta el pensamiento estructurado para resolver problemas complejos.

## 7. Bibliografía


- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
- Documentación de Python: https://docs.python.org/3/tutorial/
- Visualgo: https://visualgo.net/en/bst


## 8. Anexos

- Capturas de pantalla del código funcionando en consola.
- Repositorio en GitHub: https://github.com/grupo-arboles/estructura-arbol-python
- Video explicativo mostrando la estructura del árbol, los recorridos y reflexión final.

## 9. Entregables

- Informe (PDF) con todas las secciones anteriores.
- Script Python: `arbol_listas.py / utilidades.py / main.py`.
- Capturas de ejecución y presentación en PowerPoint.