Trabajo Integrador – Programación I

Alumnos:
Kenyi Meza (mezakenyi@gmail.com)
Méndez Roberto (robermen02@gmail.com)
Materia: Programación I
Profesor: Julieta Trapé
Fecha de entrega: 09/06/2025
Título del trabajo: Implementación de Árboles Binarios en Python Usando Listas

---

1. Introducción

Las estructuras de datos son fundamentales en el desarrollo de software, ya que permiten organizar y procesar información de manera eficiente. En este trabajo se aborda la implementación de árboles binarios utilizando exclusivamente listas en Python, sin recurrir a clases ni objetos. Esta elección facilita la comprensión estructural del árbol binario.

El objetivo principal es demostrar que, con herramientas simples como listas anidadas, es posible construir, recorrer y visualizar un árbol binario funcional. A su vez, este enfoque permite ejercitar conceptos claves como la recursividad, el diseño de funciones y la representación jerárquica de datos.

---

2. Marco Teórico

¿Qué es un árbol?

Un árbol es una estructura de datos no lineal compuesta por nodos conectados jerárquicamente. El nodo superior se denomina raíz, y cada nodo puede tener cero o más hijos. En el caso del árbol binario, cada nodo puede tener a lo sumo dos hijos: uno izquierdo y uno derecho.

Elementos importantes:

- Raíz: nodo principal.
- Hojas: nodos sin hijos.
- Altura: número de niveles del árbol.
- Subárbol: cualquier nodo con sus descendientes.

Recorridos clásicos de árboles binarios:

- Inorden: izquierda → raíz → derecha.
- Preorden: raíz → izquierda → derecha.
- Postorden: izquierda → derecha → raíz.

Representación con listas en Python

En vez de usar clases, se representa cada nodo como una lista de tres elementos:

"[nodo, subárbol_izquierdo, subárbol_derecho]"

Esto permite trabajar con estructuras jerárquicas sin necesidad de orientación a objetos, centrando la atención en la lógica de construcción y navegación del árbol.

---

3. Caso Práctico

A continuación se presenta el código fuente del árbol binario, acompañado de comentarios explicativos.

```
# Esta funcion crea un nuevo arbol con un nodo principal (raiz)
# El nodo es una lista con el valor y dos lugares vacios para los hijos
izquierdo y derecho
def crear_arbol(valor):
    return [valor, [], []]

# Esta funcion agrega un nodo a la izquierda del nodo actual
# Si ya hay algo a la izquierda, lo baja un nivel y pone el nuevo nodo arriba
def insertar_izquierda(nodo, valor):
    if nodo[1]:
        nodo[1] = [valor, nodo[1], []]
    else:
        nodo[1] = [valor, [], []]

# Esta funcion hace lo mismo que la anterior pero para el lado derecho
def insertar_derecha(nodo, valor):
    if nodo[2]:
        nodo[2] = [valor, [], nodo[2]]
    else:
        nodo[2] = [valor, [], []]

# Esta funcion elimina un nodo (referenciado por su valor[clave]) dentro de un
arbol especifico
# Maneja tres casos:
# 1 - Nodo es hoja: Se devuelve una lista vacia.
# 2 - Nodo tiene hijo izq: El nodo a eliminar se reemplaza por su hijo/rama
izquierdo.
# 3 - Nodo tiene hijo der: El nodo a eliminar se reemplaza por su hijo/rama
derecho.
# 4 - Nodo tiene dos hijos: Se reemplaza el nodo padre por su hijo izquierdo y
se conecta el hijo
#     derecho (ahora huerfano) al hijo mas hacia la derecha del "ex" hijo
izquierdo (nuevo padre).
```

```
def eliminar_nodo(arbol, valor):
    if not arbol:
        return []

    if arbol[0] == valor:
        # Caso 1 - "Nodo es hoja"
        if not arbol[1] and not arbol[2]:
            return []
        # Caso 2 - "Nodo tiene hijo izquierdo."
        elif arbol[1] and not arbol[2]:
            return arbol[1]
        # Caso 3 - "Nodo tiene hijo derecho."
        elif not arbol[1] and arbol[2]:
            return arbol[2]
        # Caso 4 - "Nodo tiene dos pibes, caos."
        else:
            hijo_izq = arbol[1]
            hijo_der = arbol[2]

            # Se reemplaza el nodo padre por su hijo izquierdo *
            actual = hijo_izq

            # Se busca al hijo mas hacia la derecha del "ex" hijo izquierdo
(nuevo padre). *
            while actual[2]:
                actual = actual[2]
            # Se conecta el hijo derecho al subarbol u hoja mas hacia la
derecha encontrada en las lineas anteriores. *
            actual[2] = hijo_der

            return hijo_izq

    # Búsqueda recursiva
    arbol[1] = eliminar_nodo(arbol[1], valor)
    arbol[2] = eliminar_nodo(arbol[2], valor)
    return arbol

# Recorre el arbol en preorden: primero muestra el nodo, despues va a la
izquierda y despues a la derecha
def recorrido_preorden(nodo):
    if nodo:
        print(nodo[0], end=' ')
        recorrido_preorden(nodo[1])
        recorrido_preorden(nodo[2])

# Recorre el arbol en inorden: primero va a la izquierda, luego muestra el
nodo, y despues va a la derecha
def recorrido_inorden(nodo):
    if nodo:
        recorrido_inorden(nodo[1])
```

```
        print(nodo[0], end=' ')
        recorrido_inorden(nodo[2])

# Recorre el arbol en postorden: primero izquierda, luego derecha, y al final
el nodo
def recorrido_postorden(nodo):
    if nodo:
        recorrido_postorden(nodo[1])
        recorrido_postorden(nodo[2])
        print(nodo[0], end=' ')

# Esta funcion muestra el arbol rotado 90 grados hacia la izquierda
def imprimir_arbol(nodo, nivel=0):
    if nodo:
        imprimir_arbol(nodo[2], nivel + 1)
        print('    ' * nivel + str(nodo[0]))
        imprimir_arbol(nodo[1], nivel + 1)

#Esta funcion busca un valor en el arbol
def buscar_valor(nodo, valor):
    if not nodo:
        return False
    if nodo[0] == valor:
        return True
    return buscar_valor(nodo[1], valor) or buscar_valor(nodo[2], valor)

#Esta funcion busca calcular la altura del arbol
def altura(nodo):
    if not nodo:
        return 0
    return 1 + max(altura(nodo[1]), altura(nodo[2]))

#Esta funcion cuenta la cantidad total de nodos
def contar_nodos(nodo):
    if not nodo:
        return 0
    return 1 + contar_nodos(nodo[1]) + contar_nodos(nodo[2])

#Esta funcion cuenta solo los nodos hoja
def mostrar_hojas(nodo):
    if nodo:
        if not nodo[1] and not nodo[2]:
            print(nodo[0], end=' ')
        mostrar_hojas(nodo[1])
        mostrar_hojas(nodo[2])


# aca armamos el arbol y probamos las funciones
if __name__ == "__main__":
    # Creamos el nodo raiz con valor 'A'
```

```python
arbol = crear_arbol('A')

# Agregamos hijos a izquierda y derecha
insertar_izquierda(arbol, 'B')
insertar_derecha(arbol, 'C')

# Agregamos nodos al segundo nivel
insertar_izquierda(arbol[1], 'D')
insertar_derecha(arbol[1], 'E')
insertar_izquierda(arbol[2], 'F')
insertar_derecha(arbol[2], 'G')

# Mostramos los recorridos
print("Recorrido Preorden:")
recorrido_preorden(arbol)

print("\nRecorrido Inorden:")
recorrido_inorden(arbol)

print("\nRecorrido Postorden:")
recorrido_postorden(arbol)

# Mostramos el arbol rotado
print("\n\nVisualizacion del arbol rotado:")
imprimir_arbol(arbol)

#Mostramos la altura
print("\n\nAltura del árbol:", altura(arbol))

#Mostramos el total de nodos
print("Cantidad total de nodos:", contar_nodos(arbol))

#Mostramos la busqueda de un nodo
print("¿Existe el nodo 'E'?:", buscar_valor(arbol, 'E'))

#Mostramos los nodo hoja
print("Nodos hoja:", end=' ')
mostrar_hojas(arbol)

print("\n\nEliminando 'B':")
arbol = eliminar_nodo(arbol, 'B')
imprimir_arbol(arbol)


# Caso práctico: Chat de diagnostico simple.
print("\n\n---- ÁRBOL DE DIAGNÓSTICO ----")

# Creamos el nodo raíz con la primera pregunta
diagnostico = crear_arbol("¿Tenés fiebre?")
```

```
    # Rama izquierda = sí
    insertar_izquierda(diagnostico, "¿Tenés dolor muscular?")
    insertar_izquierda(diagnostico[1], "Parece una gripe")
    insertar_derecha(diagnostico[1], "Parece un resfrío")

    # Rama derecha = no
    insertar_derecha(diagnostico, "¿Tenés congestión nasal?")
    insertar_izquierda(diagnostico[2], "Parece un resfrío")
    insertar_derecha(diagnostico[2], "Estás bien")

    # definimos el comportamiento del "chat" de diagnostico
    def diagnosticar(nodo):
        while nodo:
            pregunta = nodo[0]
            if not nodo[1] and not nodo[2]: # Cuando el nodo/rama es un hoja:
                # Mostramos el diagnostico final
                print("Diagnóstico:", pregunta)
                return
            respuesta = input(f"{pregunta} (sí/no): ").strip().lower()
            # Evaluamos respuesta del usuario con una validacion basica
            if respuesta in ["sí", "si"]:
                nodo = nodo[1]
            else:
                nodo = nodo[2]

    diagnosticar(diagnostico)

    print()
```

Ejecución esperada:

```
Recorrido Preorden:
A B D E C F G
Recorrido Inorden:
D B E A F C G
Recorrido Postorden:
D E B F G C A

Visualizacion del arbol rotado:
      G
   C
      F
A
      E
   B
      D


Altura del árbol: 3
```

```
Cantidad total de nodos: 7
¿Existe el nodo 'E'?: True
Nodos hoja: D E F G

Eliminando 'B':
      G
   C
      F
A
      E
   D


---- ÁRBOL DE DIAGNÓSTICO ----
¿Tenés fiebre? (sí/no): si
¿Tenés dolor muscular? (sí/no): no
Diagnóstico: Parece un resfrío
```

4. Metodología Utilizada

- Estudio del material de clase provisto por la cátedra (presentaciones y rúbricas).
- Implementación paso a paso en Python 3.11 bajo debian12.
- Desarrollo de funciones recursivas para construcción y recorridos.
- Prueba del código en vscode.
- Publicación del código en GitHub.

---

5. Resultados Obtenidos

- El árbol binario fue construido correctamente usando listas.
- Se ejecutaron con éxito los recorridos inorden, preorden y postorden.
- La visualización rotada permitió observar la jerarquía del árbol de forma clara.
- El código funciona correctamente y está documentado con comentarios en cada función.
- Se vio un ejemplo practico de para qué pueden servir los árboles como estructuras de datos. En el presente caso se utilizó para recrear una especie de “chatbot” que diagnostica y recorre el árbol según las respuestas del usuario. Si bien las aplicaciones de los árboles binarios en el plano de la realidad por lo general utilizan árboles de búsqueda binaria (que difieren un poco a nivel comportamiento/propiedades de los aquí tratados.), este pequeño programa emplea los arboles binarios como listas con ánimos de “bajar a tierra” el concepto y ver que tipo de aplicaciones puede llegar a tener el mismo.

---

6. Conclusiones

El desarrollo de este trabajo permitió aplicar la lógica de estructuras de datos en un entorno práctico, reforzando la comprensión de la recursividad y las jerarquías. El uso de listas en lugar de clases facilitó el enfoque didáctico del problema.

Este enfoque es ideal para quienes se inician en programación, ya que reduce la complejidad sintáctica y permite concentrarse en la lógica estructural. Como extensión futura, se podrían agregar funciones de búsqueda, balanceo o bien aplicar estas operaciones en los ya mencionados árboles de búsqueda binaria o BSTs (Binary Search Trees).

---

7. Bibliografía

- Material de clase.
- YouTube: https://www.youtube.com/watch?v=d0ibZK_6Q7g
- Agarwal, B. (2018) Hands-On Data Structures and Algorithms with Python.

---

8. Anexos

- Enlace al repositorio de GitHub: https://github.com/ya-awn/arboles-en-python
- Video explicativo : https://youtu.be/EYz3A9IDjlg

```
PS C:\Users\Usuario\Desktop\others\arboles-en-python\python> python arbol_listas.py
Recorrido Preorden:
A B D E C F G
Recorrido Inorden:
D B E A F C G
Recorrido Postorden:
D E B F G C A

Visualizacion del arbol rotado:
      G
   C
      F
A
      E
   B
      D


Altura del árbol: 3
Cantidad total de nodos: 7
¿Existe el nodo 'E'?: True
Nodos hoja: D E F G

Eliminando 'B':
      G
   C
      F
A
      E
   D


---- ÁRBOL DE DIAGNÓSTICO ----
¿Tenés fiebre? (sí/no): Si
¿Tenés dolor muscular? (sí/no): si
Diagnóstico: Parece una gripe
```