UNIVERSIDAD TECNOLOGICA NACIONAL

Trabajo Práctico Integrador

Investigación aplicada en Python - Estructuras de Datos Avanzadas.

Arboles Binarios

Alumnos

Aguirre Luciano & Zupan Lucas

Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional.

Programacion I

Docente Titular

Nicolás Quirós

Docente Tutor

Giuliano Crenna

09 de Junio de 2025

1. Introducción

El presente trabajo se centra en el estudio e implementación de **estructuras de datos avanzadas**, con especial foco en los **árboles binarios** y otros tipos de árboles. Este tema fue elegido debido a su relevancia tanto teórica como práctica en el campo de la programación, y por su amplia presencia en áreas fundamentales de la informática como bases de datos, algoritmos de búsqueda, inteligencia artificial y sistemas de archivos. Los árboles permiten representar y gestionar datos jerárquicos de manera eficiente, lo que los convierte en herramientas clave para resolver problemas complejos en distintos contextos.

En el ámbito de las estructuras de datos, los árboles ocupan un lugar destacado por su capacidad de optimizar operaciones fundamentales como búsqueda, inserción y eliminación. Estructuras como los árboles binarios de búsqueda (BST), árboles AVL, B-Trees, entre otros, ofrecen ventajas significativas en términos de rendimiento y escalabilidad. Por ello, conocer su funcionamiento e implementación resulta esencial para cualquier desarrollador o profesional en ciencias de la computación.

El objetivo de este trabajo es investigar, implementar y analizar el funcionamiento de los árboles binarios mediante el desarrollo de un caso práctico utilizando el lenguaje Python. Se busca integrar los conceptos teóricos con la práctica, explorando cómo se construye, recorre y gestiona esta estructura de datos. A través de esta implementación, se pretende reforzar los conocimientos adquiridos en la materia y demostrar la aplicabilidad de los árboles binarios en contextos reales de programación.

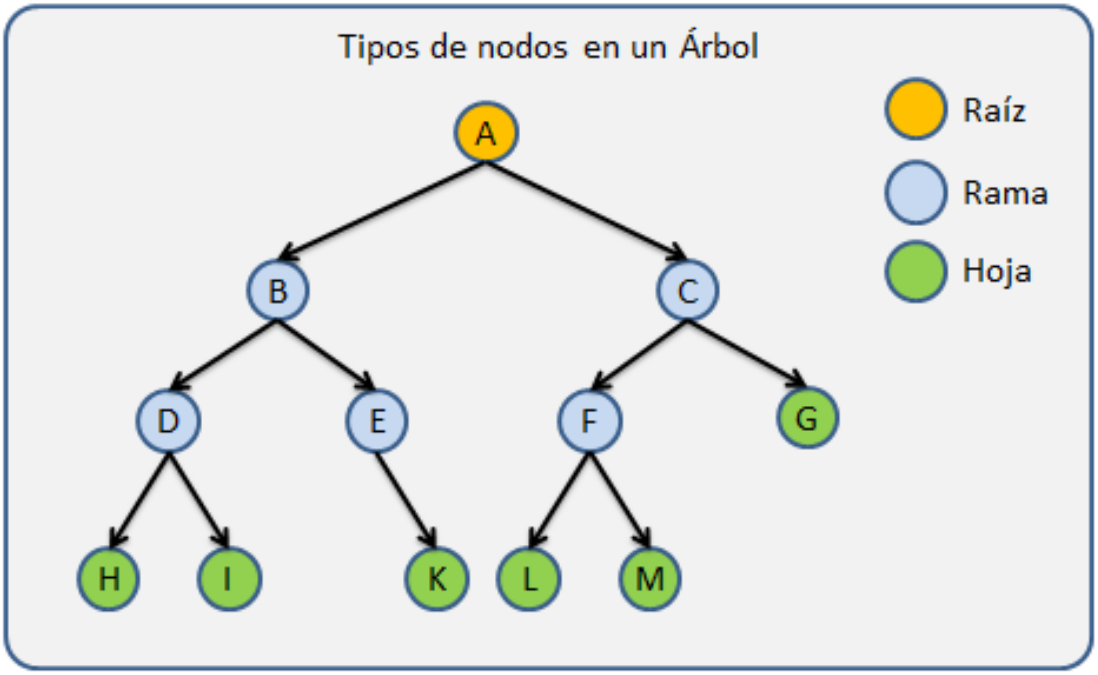
2. Marco Teórico

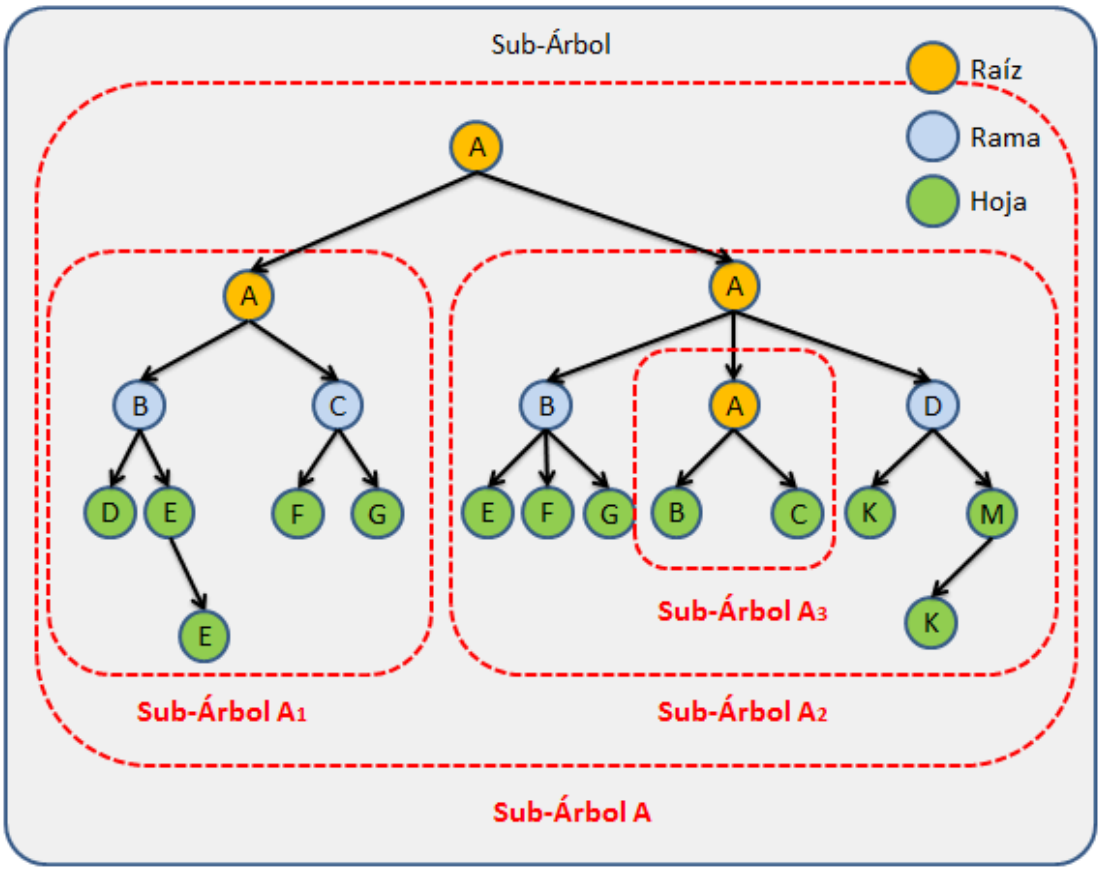
Definición de Árboles:

En ciencias de la computación, un **árbol** es una estructura de datos no lineal que organiza elementos de forma jerárquica. Está compuesto por **nodos**, donde cada nodo puede tener cero o más **hijos**, pero solo un **padre**, excepto el nodo raíz, que no tiene ninguno. Esta estructura se utiliza ampliamente para representar relaciones jerárquicas, como los sistemas de archivos, árboles de decisión, árboles genealógicos o expresiones algebraicas.

Componentes y terminología:

- **Raíz**: Nodo principal del árbol (sin padre).
- **Nodo**: Unidad básica que contiene un valor y referencias a otros nodos.
- **Hijo**: Nodo descendiente de otro nodo.
- **Padre**: Nodo que tiene al menos un hijo.
- **Hoja**: Nodo sin hijos.
- **Subárbol**: Conjunto de nodos descendientes de un nodo.
- **Altura**: Longitud del camino más largo desde la raíz hasta una hoja.
- **Nivel**: Profundidad de un nodo respecto a la raíz.
- **Balanceo**: Mecanismo para mantener el árbol equilibrado en altura.


Tipos de nodos en un Árbol
Raíz
Rama
Hoja
A
B
C
D
E
F
G
H
I
K
L
M


Sub-Árbol
Raíz
Rama
Hoja
Sub-Árbol A1
Sub-Árbol A2
Sub-Árbol A3
Sub-Árbol A

Recorridos en Árboles Binarios

Los **recorridos** permiten visitar sistemáticamente todos los nodos de un árbol. En los árboles binarios, existen tres formas principales de recorrerlos:

1. Recorrido Inorden (Izquierda - Raíz - Derecha)

Se recorre primero el subárbol izquierdo, luego el nodo actual (raíz) y finalmente el subárbol derecho. En un **Árbol Binario de Búsqueda (BST)**, este recorrido devuelve los valores en orden ascendente.

2. Recorrido Preorden (Raíz - Izquierda - Derecha)

Primero se procesa el nodo actual, luego el subárbol izquierdo y finalmente el derecho. Se utiliza comúnmente para clonar árboles o generar expresiones prefijadas.

3. Recorrido Postorden (Izquierda - Derecha - Raíz)

Se recorre primero el subárbol izquierdo, luego el derecho y por último el nodo actual. Es útil para eliminar un árbol o evaluar expresiones.

Clasificación de Árboles:

A continuación se detallan los principales tipos de árboles utilizados en programación:

Tipo de Árboles	**Características principales**	**Uso frecuente**
Árbol Binario	Cada nodo tiene hasta dos hijos (izquierdo y derecho)	Estructuras base en algoritmos y análisis
Árbol Binario de Búsqueda (BST)	Los nodos a la izquierda son menores; los de la derecha, mayores	Búsqueda e inserción ordenada de datos
Árbol AVL	BST auto-balanceado; mantiene equilibrada su altura mediante rotaciones	Optimización de operaciones de búsqueda
Heap Binario	Árbol binario completo que mantiene un orden específico entre nodos	Estructura base para colas de prioridad
Árbol B / B+	Generalizan los árboles binarios para manejar grandes cantidades de datos	Sistemas de archivos y bases de datos

Implementación en Python:

En Python, los árboles no están incluidos como una estructura estándar (como listas o diccionarios), pero se pueden implementar fácilmente usando clases. A continuación se presenta un ejemplo básico de un **árbol binario de búsqueda (BST)**.

3. Caso Práctico

Descripción del problema

Para aplicar los conceptos teóricos trabajados, se desarrolló una simulación de un **Árbol Binario de Búsqueda** (BST) en Python, con el objetivo de almacenar, organizar y recorrer un conjunto de números de forma eficiente.

El problema simula un sistema que necesita mantener una colección de datos numéricos ordenados para permitir su recorrido de distintas maneras. Este tipo de operación es común, por ejemplo, en sistemas de archivos, catálogos ordenados o estructuras jerárquicas de toma de decisiones.

Decisiones de diseño

El desarrollo del sistema se centró en construir un Árbol Binario de Búsqueda (BST) completamente funcional, con soporte para inserción, eliminación y recorridos clásicos (inorden, preorden y postorden), incluyendo visualización dinámica utilizando la librería Graphviz.

Se diseñó una estructura orientada a objetos que incluye dos clases principales: `Nodo` y `ArbolBST`. La clase `Nodo` representa cada elemento del árbol con referencias a sus hijos izquierdo y derecho. La clase `ArbolBST` contiene todos los métodos necesarios para operar sobre el árbol, manteniendo encapsulada la lógica recursiva de inserción, eliminación y recorridos.

Para facilitar la comprensión visual del funcionamiento interno del árbol, se integró la biblioteca `graphviz`, que permite generar gráficos automáticos del árbol en distintos estados. Se implementó una función que genera no solo un gráfico básico del árbol, sino también representaciones diferenciadas según los recorridos (preorden, inorden, postorden), incluyendo colores personalizados y etiquetas con el orden de visita de cada nodo.

Se incorporó además un menú interactivo que permite al usuario:

- Insertar y eliminar múltiples valores.
- Generar árboles aleatorios dentro de un rango configurable.
- Mostrar recorridos textuales.
- Generar automáticamente todas las visualizaciones gráficas del árbol.

Validación del funcionamiento

Se validó el funcionamiento del árbol verificando manualmente que los recorridos generarán los resultados esperados:

- El recorrido **inorden** muestra los datos ordenados ascendentemente.
- El **preorden** y **postorden** coinciden con los patrones estructurales definidos en teoría.
- El árbol respeta la propiedad del BST: todos los valores a la izquierda son menores, y a la derecha, mayores.

Esto demuestra que la estructura y sus métodos fueron correctamente implementados y se comportan de forma esperada frente a distintos conjuntos de datos.

Código del programa en Python

```
from graphviz import Digraph
import random


class Nodo:
    def __init__(self, valor):
        self.valor = valor
        self.izq = None
        self.der = None


class ArbolBST:
    def __init__(self):
        self.raiz = None

    def insertar(self, valor):
        self.raiz = self._insertar_recursivo(self.raiz, valor)

    def _insertar_recursivo(self, nodo, valor):
        if nodo is None:
            return Nodo(valor)
        if valor < nodo.valor:
            nodo.izq = self._insertar_recursivo(nodo.izq, valor)
        elif valor > nodo.valor:
            nodo.der = self._insertar_recursivo(nodo.der, valor)
        return nodo  # No inserta duplicados

    def eliminar(self, valor):
        self.raiz = self._eliminar_recursivo(self.raiz, valor)
```

```
    def _eliminar_recursivo(self, nodo, valor):
        if nodo is None:
            return None
        if valor < nodo.valor:
            nodo.izq = self._eliminar_recursivo(nodo.izq, valor)
        elif valor > nodo.valor:
            nodo.der = self._eliminar_recursivo(nodo.der, valor)
        else:
            # Nodo encontrado: 3 casos
            if nodo.izq is None:
                return nodo.der
            elif nodo.der is None:
                return nodo.izq
            # Nodo con dos hijos: reemplazar por el menor del subárbol derecho
            temp = self._min_valor_nodo(nodo.der)
            nodo.valor = temp.valor
            nodo.der = self._eliminar_recursivo(nodo.der, temp.valor)
        return nodo

    def _min_valor_nodo(self, nodo):
        actual = nodo
        while actual.izq:
            actual = actual.izq
        return actual

    # Recorridos del árbol
    def preorder(self):
        resultado = []
        self._preorder_recursivo(self.raiz, resultado)
        return resultado

    def _preorder_recursivo(self, nodo, resultado):
        if nodo:
            resultado.append(nodo.valor)
            self._preorder_recursivo(nodo.izq, resultado)
            self._preorder_recursivo(nodo.der, resultado)

    def inorder(self):
        resultado = []
        self._inorder_recursivo(self.raiz, resultado)
        return resultado

    def _inorder_recursivo(self, nodo, resultado):
        if nodo:
            self._inorder_recursivo(nodo.izq, resultado)
```

```
            resultado.append(nodo.valor)
            self._inorder_recursivo(nodo.der, resultado)

    def postorder(self):
        resultado = []
        self._postorder_recursivo(self.raiz, resultado)
        return resultado

    def _postorder_recursivo(self, nodo, resultado):
        if nodo:
            self._postorder_recursivo(nodo.izq, resultado)
            self._postorder_recursivo(nodo.der, resultado)
            resultado.append(nodo.valor)

    def generar_grafico(self, nombre_archivo="arbol_interactivo",
mostrar_recorrido=None):
        if self.raiz is None:
            print("El árbol está vacío. No se puede generar el gráfico.")
            return

        dot = Digraph(format='png')
        dot.attr(rankdir='TB', fontname='Arial', size='12,8')

        # Obtener el orden del recorrido si se especifica
        orden_recorrido = {}
        if mostrar_recorrido:
            if mostrar_recorrido == 'preorder':
                recorrido = self.preorder()
                titulo = "Preorder (Raíz → Izq → Der)"
            elif mostrar_recorrido == 'inorder':
                recorrido = self.inorder()
                titulo = "Inorder (Izq → Raíz → Der)"
            elif mostrar_recorrido == 'postorder':
                recorrido = self.postorder()
                titulo = "Postorder (Izq → Der → Raíz)"

            # Crear diccionario con el orden de visita
            for i, valor in enumerate(recorrido, 1):
                orden_recorrido[valor] = i

            # Agregar título al gráfico
            dot.attr(label=f'\\n{titulo}\\nSecuencia: {" → ".join(map(str,
recorrido))}',
                     labelloc='t', fontsize='14')
```

```
        self._agregar_nodos(dot, self.raiz, es_raiz=True,
orden_recorrido=orden_recorrido)

        # Generar nombre de archivo apropiado
        if mostrar_recorrido:
            nombre_completo = f"{nombre_archivo}_{mostrar_recorrido}"
        else:
            nombre_completo = nombre_archivo

        dot.render(nombre_completo, cleanup=True)
        print(f"Imagen generada: {nombre_completo}.png")


    def generar_todos_los_graficos(self, nombre_base="arbol"):
        if self.raiz is None:
            print("El árbol está vacío. No se pueden generar los gráficos.")
            return

        print("Generando todos los gráficos...")

        # Gráfico básico (sin recorrido)
        self.generar_grafico(nombre_base)

        # Gráficos con recorridos
        recorridos = ['preorder', 'inorder', 'postorder']
        for recorrido in recorridos:
            self.generar_grafico(nombre_base, recorrido)

        print(f"¡Completado! Se generaron 4 archivos:")
        print(f"   • {nombre_base}.png (árbol básico)")
        print(f"   • {nombre_base}_preorder.png")
        print(f"   • {nombre_base}_inorder.png")
        print(f"   • {nombre_base}_postorder.png")


    def _agregar_nodos(self, dot, nodo, es_raiz=False, orden_recorrido=None):
        if nodo is None:
            return

        # Determinar color del nodo
        if es_raiz:
            color = 'lightgreen'
        elif nodo.izq is None and nodo.der is None:
            color = 'lightblue'
        else:
```

```
            color = 'lightgray'

        # Crear etiqueta del nodo
        if orden_recorrido and nodo.valor in orden_recorrido:
            # Mostrar valor y orden de recorrido
            label = f"{nodo.valor}\\n({orden_recorrido[nodo.valor]})"
            # Usar colores más intensos para destacar el recorrido
            if es_raiz:
                color = 'green'
            elif nodo.izq is None and nodo.der is None:
                color = 'deepskyblue'
            else:
                color = 'gray'
        else:
            label = str(nodo.valor)

        dot.node(
            str(nodo.valor),
            label=label,
            style='filled',
            fillcolor=color,
            shape='ellipse',
            fontname='Arial',
            fontsize='12'
        )

        # Agregar conexiones
        if nodo.izq:
            self._agregar_nodos(dot, nodo.izq, orden_recorrido=orden_recorrido)
            dot.edge(str(nodo.valor), str(nodo.izq.valor))
        if nodo.der:
            self._agregar_nodos(dot, nodo.der, orden_recorrido=orden_recorrido)
            dot.edge(str(nodo.valor), str(nodo.der.valor))

    def generar_numeros_aleatorios(self, cantidad):
        """Genera números aleatorios según la cantidad especificada"""
        # Limpiar el árbol actual
        self.raiz = None

        # Generar números aleatorios únicos
        numeros_generados = set()

        # Rango de números aleatorios (más amplio para evitar colisiones)
        rango_min = 1
        rango_max = cantidad * 10  # Rango amplio para evitar duplicados
```

```
        while len(numeros_generados) < cantidad:
            numero = random.randint(rango_min, rango_max)
            numeros_generados.add(numero)

        # Convertir a lista y mezclar para orden de inserción aleatorio
        lista_numeros = list(numeros_generados)
        random.shuffle(lista_numeros)

        # Insertar los números en el árbol
        for numero in lista_numeros:
            self.insertar(numero)

        print(f"Números generados aleatoriamente: {sorted(lista_numeros)}")
        print(f"Total de nodos: {len(lista_numeros)}")
        print(f"Orden de inserción: {lista_numeros}")

        return lista_numeros

    def mostrar_recorridos(self):
        if self.raiz is None:
            print("El árbol está vacío.")
            return

        print("\nRECORRIDOS DEL ÁRBOL:")
        print(f"Preorder:  {' → '.join(map(str, self.preorder()))}")
        print(f"Inorder:   {' → '.join(map(str, self.inorder()))}")
        print(f"Postorder: {' → '.join(map(str, self.postorder()))}")

# Menú Interactivo
def menu():
    arbol = ArbolBST()
    while True:
        print("\n MENÚ:")
        print("1. Insertar valores (separados por coma)")
        print("2. Eliminar valores (separados por coma)")
        print("3. Generar árbol con números aleatorios")
        print("4. Generar TODOS los gráficos (básico + recorridos)")
        print("5. Mostrar recorridos (texto)")
        print("6. Salir")
        opcion = input("Elegí una opción: ")


        if opcion == '1':
            entrada = input(" Ingresá valores a insertar (ej: 10,20,30): ")
```

```
            try:
                valores = [int(v.strip()) for v in entrada.split(",")]
                for v in valores:
                    arbol.insertar(v)
                print("Valores insertados.")
            except ValueError:
                print("Entrada inválida. Usá solo números separados por coma.")

        elif opcion == '2':
            entrada = input("🗑️ Ingresá valores a eliminar (ej: 10,20): ")
            try:
                valores = [int(v.strip()) for v in entrada.split(",")]
                for v in valores:
                    arbol.eliminar(v)
                print("Valores eliminados (si existían).")
            except ValueError:
                print("Entrada inválida. Usá solo números separados por coma.")

        elif opcion == '3':
            while True:
                try:
                    cantidad = input("¿Cuántos nodos querés en el árbol? (3-50): ")
                    cantidad = int(cantidad)

                    if cantidad < 3:
                        print("Mínimo 3 nodos para tener un árbol interesante.")
                        continue

                    elif cantidad > 50:
                        print("Máximo 50 nodos para mantener legible el gráfico.")
                        continue

                    else:
                        arbol.generar_numeros_aleatorios(cantidad)
                        break

                except ValueError:
                    print("Por favor ingresá un número válido.")
        elif opcion == '4':
            arbol.generar_todos_los_graficos()

        elif opcion == '5':
            arbol.mostrar_recorridos()

        elif opcion == '6':
```

```
            print("Programa finalizado.")
            break

        else:
            print("Opción no válida. Intentá de nuevo.")
# Ejecutar
if __name__ == "__main__":
    menu()
```

4. Metodología Utilizada

El desarrollo del trabajo se llevó a cabo en varias etapas, combinando investigación teórica, diseño de estructuras y programación práctica en Python. A continuación se describen los pasos seguidos:

Investigación previa

Se realizó una revisión de conceptos fundamentales sobre árboles binarios, árboles binarios de búsqueda (BST) y sus recorridos clásicos (`inorden`, `preorden` y `postorden`).

Diseño e implementación del código:

Se construyó un árbol BST desde cero en Python, aplicando técnicas de recursividad para la inserción, eliminación y recorridos.

A lo largo del desarrollo se llevaron a cabo pruebas incrementales, testeando cada método de forma aislada antes de integrarlo al sistema completo.

Se priorizó un diseño modular para facilitar la lectura del código y permitir su extensión.

Visualización del árbol:

Se incorporó la librería `Graphviz` para representar gráficamente el árbol y resaltar los recorridos. Esta decisión permitió una mejor comprensión visual de la estructura y facilitó la validación de su funcionamiento interno.

Desarrollo de interfaz por consola:

Se diseñó un menú interactivo para que el usuario pueda operar con el árbol de forma sencilla: insertar o eliminar valores, generar árboles aleatorios, visualizar recorridos y exportar gráficos.

Esta funcionalidad hizo que el programa sea más accesible, ideal para propósitos educativos o de demostración.

Herramientas utilizadas

- **Lenguaje:** Python 3.12
- **IDE:** Visual Studio Code
- **Control de versiones:** Git (repositorio local)
- **Librerías auxiliares:** `graphviz` para visualización, `random` para generación de datos.
- **Sistemas operativos utilizados:** Windows y Linux (Ubuntu)

5. Resultados Obtenidos

El caso práctico permitió implementar con éxito un Árbol Binario de Búsqueda (BST) completamente funcional, interactivo y visualizable. A continuación se detallan los principales resultados alcanzados, dificultades encontradas y pruebas realizadas.

Funcionalidades logradas

- **Inserción dinámica** de múltiples valores mediante consola.
- **Eliminación controlada** de nodos existentes.
- **Visualización automática** del árbol mediante la librería `graphviz`, con estilo mejorado y colores que diferencian la raíz, los nodos hoja e internos.
- **Menú interactivo** que permite ejecutar todas las operaciones desde una interfaz simple en consola.

Casos de prueba realizados

Se realizaron varias pruebas utilizando distintos conjuntos de valores, incluyendo:

Caso de prueba	Resultado esperado	Observaciones
Insertar `[50,30,70]`	Árbol con raíz 50, hijos 30 (izq), 70 (der)	Visualización correcta
Insertar duplicados `30`	Ignorado (no se duplica el nodo)	Comportamiento esperado
Eliminar nodo hoja `70`	Se elimina sin afectar otros nodos	Correcto
Eliminar nodo con hijos `30`	Reemplazo correcto por sucesor inorden	Árbol mantiene estructura BST
Insertar gran cantidad de nodos de manera aleatoria	Genera árbol complejo y equilibrado visualmente	Generación del gráfico exitosa

En la sección Anexos se encuentran las capturas de estos casos de prueba

Dificultades encontradas

- **Control de duplicados en el árbol**:
  Se debió modificar la lógica de inserción para evitar la carga de valores repetidos, lo que afectaría la propiedad del BST.

- **Manejo de eliminación de nodos con dos hijos**:
  Fue necesario aplicar una lógica específica que reemplaza el nodo por el menor valor del subárbol derecho, lo cual implicó una comprensión más profunda del algoritmo.

- **Visualización adecuada del árbol**:
  La primera versión generaba gráficos poco claros o desordenados. Se solucionó ajustando etiquetas, colores y espaciado para mejorar la legibilidad.

- **Recorridos mal visualizados**:
  Hubo errores al representar el orden de los recorridos (pre, in y postorden). Se resolvió mostrando los pasos directamente en los nodos mediante anotaciones y colores.

- **Dependencia de la librería Graphviz**:
  Al principio no se generaban imágenes por falta de instalación del ejecutable de Graphviz (no basta con el módulo de Python). Se corrigió instalando correctamente tanto el paquete como el binario.

- **Errores de entrada del usuario**:
  Se detectaron fallos al ingresar datos no numéricos o mal formateados. Se agregó validación por consola para prevenir estos errores.

- **Elección del tamaño del árbol aleatorio**:
  Si se generaban muchos nodos, el gráfico resultaba ilegible. Se limitó el tamaño entre 3 y 50 elementos para asegurar claridad.

Evaluación de rendimiento (básica)

Si bien el objetivo no fue medir rendimiento en detalle, se observó que incluso con **30+ nodos** el programa responde de forma fluida, tanto para inserciones como para generación del gráfico. Las operaciones básicas (insertar, eliminar) se mantienen en tiempo O(log n) en el mejor de los casos, propio de un BST balanceado.

6. Conclusiones

A lo largo del desarrollo del trabajo, el grupo pudo profundizar en el funcionamiento de los árboles binarios como estructura de datos fundamental en programación. Se aprendió a implementar desde cero un árbol binario de búsqueda (BST) en Python, comprendiendo no solo su lógica interna de inserción, eliminación y recorrido, sino también la importancia de mantener una estructura coherente y clara para optimizar su funcionamiento.

El tema resultó especialmente útil para reforzar conceptos vinculados a la recursividad, la representación jerárquica de datos y la abstracción de estructuras complejas. Además, permitió

aplicar herramientas adicionales como `graphviz` para visualizar de forma efectiva la estructura construida, facilitando la comprensión y depuración del código.

Entre las principales dificultades, se destacaron los desafíos técnicos asociados a la instalación y configuración de librerías externas, así como la generación de visualizaciones legibles en árboles con muchos nodos. También se presentaron complicaciones al manejar entradas del usuario y al implementar operaciones como la eliminación de nodos con múltiples hijos. Cada uno de estos obstáculos fue abordado con soluciones progresivas, validaciones de entrada, y mejoras en el diseño del programa.

En resumen, el trabajo permitió consolidar conocimientos fundamentales en estructuras de datos, enfrentar problemas reales de programación, y proponer soluciones prácticas a partir de una base teórica sólida.

7. Bibliografía

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

- Python documentation: https://docs.python.org/3/

- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). *Data Structures and Algorithms in Python*. Wiley.

8. Anexos

Link al video explicativo:

LINK VIDEO INSERTAR

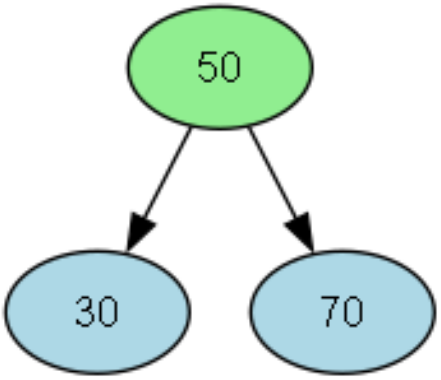
Link al repositorio:

LINK REPOSITORIO INSERTAR

Casos de prueba realizados

- Insertar `[50,30,70]`
  Se genera un árbol con los datos 50,30,70 y se muestra la imagen del resultado

```
  MENU:
1. Insertar valores (separados por coma)
2. Eliminar valores (separados por coma)
3. Generar árbol con números aleatorios
4. Generar TODOS los gráficos (básico + recorridos)
5. Mostrar recorridos (texto)
6. Salir
Elegí una opción: 1
 Ingresá valores a insertar (ej: 10,20,30): 50,30,70
Valores insertados.
```



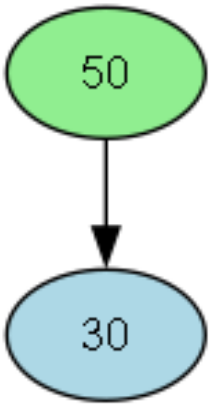
- Insertar duplicados `30`
  Partiendo del árbol del caso de prueba anterior se intenta agregar el nodo 30 y se muestra que el árbol sigue teniendo 3 nodos, lo que indica que no agrego el 30 de manera duplicada

```
6. Salir
Elegí una opción: 1
 Ingresá valores a insertar (ej: 10,20,30): 30
Valores insertados.

 MENÚ:
1. Insertar valores (separados por coma)
2. Eliminar valores (separados por coma)
3. Generar árbol con números aleatorios
4. Generar TODOS los gráficos (básico + recorridos)
5. Mostrar recorridos (texto)
6. Salir
Elegí una opción: 5

RECORRIDOS DEL ÁRBOL:
Preorder:  50 → 30 → 70
Inorder:   30 → 50 → 70
Postorder: 30 → 70 → 50
```

- Eliminar nodo hoja 70

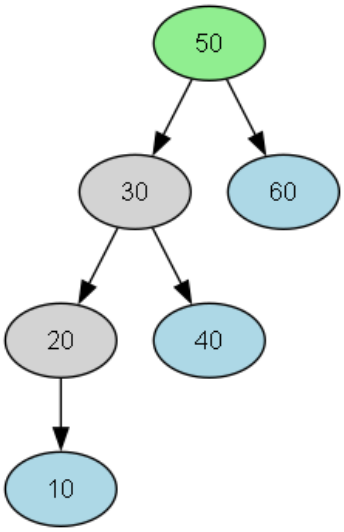
  Al árbol de los casos de prueba anteriores se le elimina el nodo 70 y se muestra el resultado.

```
 MENÚ:
1. Insertar valores (separados por coma)
2. Eliminar valores (separados por coma)
3. Generar árbol con números aleatorios
4. Generar TODOS los gráficos (básico + recorridos)
5. Mostrar recorridos (texto)
6. Salir
Elegí una opción: 2
🗑 Ingresá valores a eliminar (ej: 10,20): 70
Valores eliminados (si existían).
```



- Eliminar nodo con hijos 30

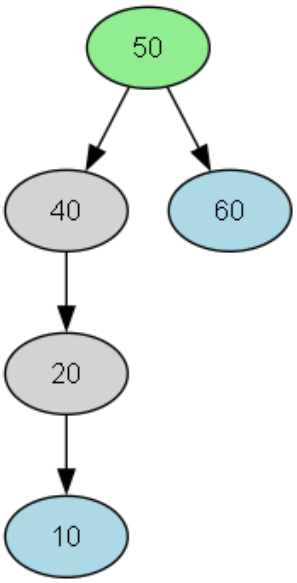
  Se parte de este árbol:



Se elimina el nodo 30:

```
 MENÚ:
1. Insertar valores (separados por coma)
2. Eliminar valores (separados por coma)
3. Generar árbol con números aleatorios
4. Generar TODOS los gráficos (básico + recorridos)
5. Mostrar recorridos (texto)
6. Salir
Elegí una opción: 2
🗑 Ingresá valores a eliminar (ej: 10,20): 30
Valores eliminados (si existían).
```

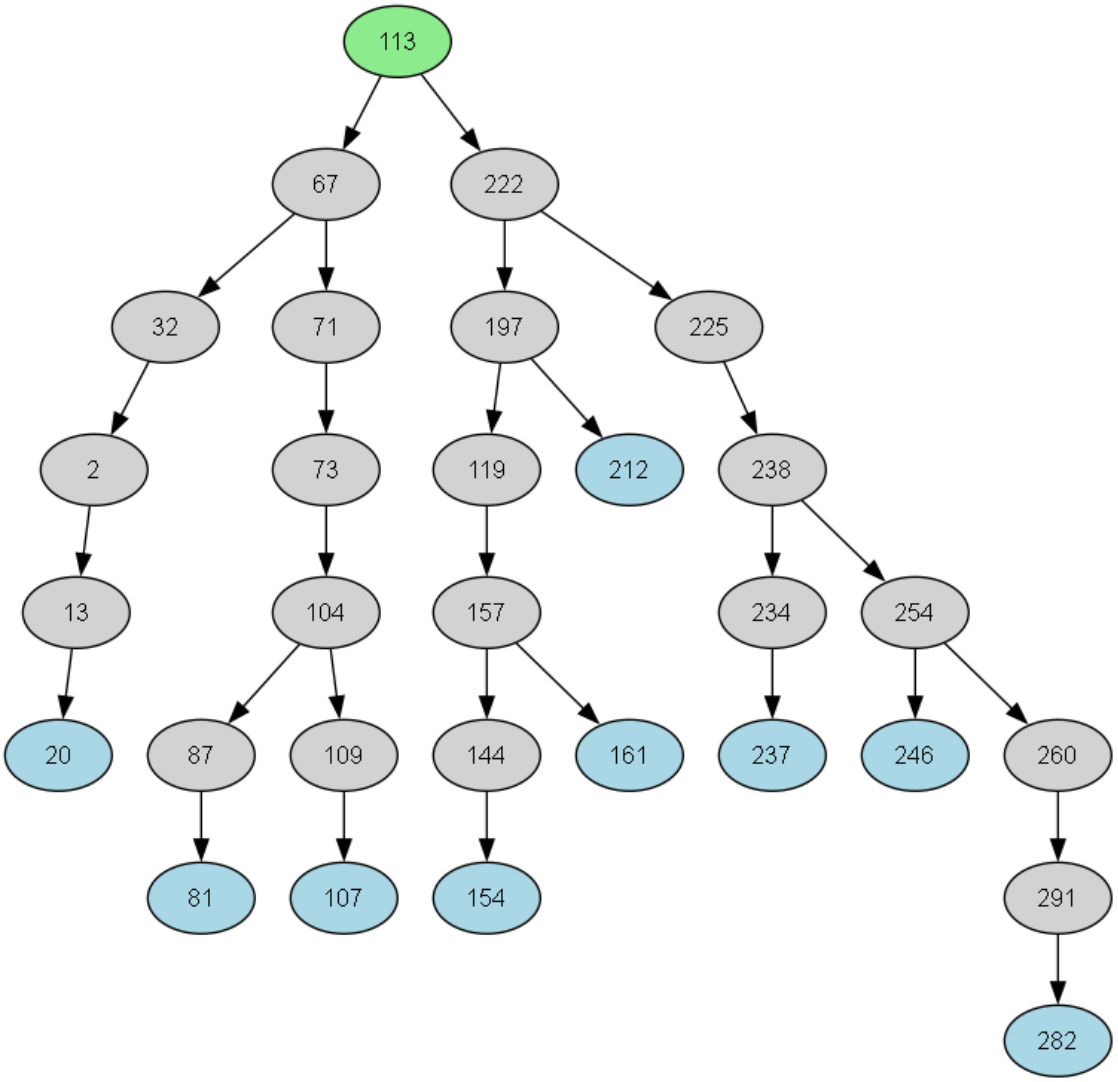
Se muestra el resultado de eliminar el nodo 30:



- Insertar gran cantidad de nodos de manera aleatoria

Se arma un árbol aleatorio con 30 nodos:

```
 MENÚ:
1. Insertar valores (separados por coma)
2. Eliminar valores (separados por coma)
3. Generar árbol con números aleatorios
4. Generar TODOS los gráficos (básico + recorridos)
5. Mostrar recorridos (texto)
6. Salir
Elegí una opción: 3
¿Cuántos nodos querés en el árbol? (3-50): 30
Números generados aleatoriamente: [2, 13, 20, 32, 67, 71, 73, 81, 87, 104, 107, 109, 113, 119, 144, 154, 157, 161, 197, 212, 222, 225, 234, 237, 238, 246, 254, 260, 28
2, 291]
Total de nodos: 30
Orden de inserción: [113, 67, 222, 197, 71, 225, 119, 238, 73, 32, 104, 254, 157, 260, 2, 291, 13, 161, 144, 154, 109, 246, 87, 107, 282, 81, 212, 20, 234, 237]
```



Material adicional que no va en el cuerpo principal del trabajo pero que aporta valor.
Pueden ser:

- Capturas del programa funcionando.
- Enlace al video explicativo.
- Código completo como archivo externo o adjunto.
- Cuadros comparativos.
- Documentos auxiliares (como diagramas de flujo).