

Actividad Evaluativa - Eje 2- Árbol

Melqui Romero y Karolaine Villero

Universidad Área Andina

Fundación Universitaria del Área Andina

Ingeniería en Sistemas

Modelos de Programación II

Ing. Deivys Morales

02 de marzo del 2025

Actividad Evaluativa - Eje 2- Árbol

En el campo de las ciencias de la computación, la organización eficiente de datos es fundamental para el desarrollo de software robusto y eficaz. Entre las diversas estructuras de datos disponibles, los árboles destacan como una de las más versátiles y poderosas, especialmente cuando se trata de representar relaciones jerárquicas y realizar operaciones de búsqueda optimizadas.

Los árboles binarios, en particular, son estructuras no lineales que organizan los datos de manera jerárquica, donde cada elemento (nodo) puede tener hasta dos hijos, permitiendo una organización natural de la información que facilita operaciones como búsqueda, inserción y eliminación de datos. Esta característica los hace especialmente útiles en aplicaciones que requieren búsquedas eficientes y manejo dinámico de datos (Cormen et al., 2022).

A diferencia de las estructuras lineales como arreglos o listas enlazadas, los árboles binarios ofrecen una complejidad logarítmica $O(\log n)$ para sus operaciones básicas cuando están balanceados, lo que los hace significativamente más eficientes para grandes conjuntos de datos. Además, su naturaleza dinámica permite que la estructura crezca o disminuya según las necesidades del programa durante su ejecución (Meneses, 2021).

Este proyecto no solo servirá como ejemplo práctico de implementación, sino también como demostración de cómo los conceptos teóricos de estructuras de datos se aplican en soluciones de programación reales.

Objetivo General

Desarrollar e implementar una estructura de datos tipo Árbol Binario en Python que demuestre la eficiencia y versatilidad del manejo de datos jerárquicos, mediante la creación de un sistema completo que incluya operaciones avanzadas de búsqueda, recorrido y análisis de nodos, permitiendo así la comprensión práctica de conceptos fundamentales de estructuras de datos no lineales y su aplicación en la resolución de problemas computacionales complejos, con énfasis en la optimización de recursos y la eficiencia algorítmica.

Objetivos Específicos

Implementar una estructura base de árbol binario que permita la gestión eficiente de datos mediante operaciones fundamentales de inserción, recorrido y búsqueda, garantizando la integridad y coherencia de la información almacenada.

Desarrollar un conjunto de algoritmos especializados para el análisis y manipulación de nodos, incluyendo la identificación de características específicas y el cálculo de propiedades relacionales entre los elementos del árbol.

Diseñar una interfaz de usuario que facilite la interacción con la estructura de datos, permitiendo la visualización clara de resultados y la ejecución intuitiva de las diferentes operaciones implementadas.

Tabla de contenido

Definición Árbol	2
Características de los Árboles Binarios.....	2
Operaciones Fundamentales.....	3
Implementación.....	5
Estructura General del Código	5
Descripción de Componentes.....	6
4.2.1. Clase Nodo	6
Clase Árbol Binario.....	7
Archivo main sep,.py	13
Principales operaciones implementadas:	13
4.2.3. Interfaz de Usuario.....	13
La interfaz de consola proporciona:	13
Estructura general del código	15
Aplicación principal – Por Espacios (APP):	15
Aplicación principal – Por Comas (APP):	16
Estructura del programa	16
Video explicativo	17
Como ejecutarlo	18
Salidas	18
Conclusiones	19
Referencias	20

Definición Árbol

Un árbol es una estructura de datos no lineal que simula una estructura jerárquica con un conjunto de nodos conectados. Cada árbol tiene las siguientes características fundamentales:

- Existe un único nodo llamado raíz que no tiene padre
- Cada nodo (excepto la raíz) tiene exactamente un nodo padre
- Cada nodo puede tener cero o más nodos hijos
- Los nodos sin hijos se denominan hojas o nodos terminales
- La relación entre los nodos forma una estructura jerárquica donde la información se organiza de manera natural, similar a un árbol genealógico o una estructura organizacional.

Características de los Árboles Binarios

Los árboles binarios son un tipo específico de árbol con las siguientes características distintivas:

Estructura:

- Cada nodo tiene como máximo dos hijos
- Los hijos se denominan "hijo izquierdo" e "hijo derecho"
- La altura del árbol es la longitud del camino más largo desde la raíz hasta una hoja

Propiedades:

- Un árbol binario de altura h puede tener entre $h+1$ y $2^{(h+1)}-1$ nodos
- En un nivel n pueden existir como máximo 2^n nodos

- Un árbol binario con n nodos tiene exactamente $n-1$ aristas

Tipos Especiales:

- Árbol binario completo: todos los niveles están llenos excepto posiblemente el último
- Árbol binario perfecto: todos los niveles están completamente llenos
- Árbol binario degenerado: cada nodo tiene como máximo un hijo

Operaciones Fundamentales

Las operaciones básicas que se pueden realizar en un árbol binario incluyen:

Operaciones de Recorrido:

- In-orden (izquierda, raíz, derecha)
- Pre-orden (raíz, izquierda, derecha)
- Post-orden (izquierda, derecha, raíz)

Complejidad temporal: $O(n)$, donde n es el número de nodos

Operaciones de Búsqueda:

- Búsqueda de un valor específico
- Búsqueda del mínimo/máximo
- Búsqueda de caminos

Complejidad temporal: $O(h)$, donde h es la altura del árbol

Operaciones de Modificación:

- Inserción de nuevos nodos

- Eliminación de nodos existentes
- Actualización de valores

Complejidad temporal: $O(h)$ en el caso promedio

Operaciones de Análisis:

- Cálculo de altura
- Conteo de nodos
- Verificación de propiedades específicas

Complejidad temporal: $O(n)$ en el peor caso

La eficiencia de estas operaciones depende significativamente de la estructura y balance del árbol. En un árbol binario de búsqueda balanceado, muchas de estas operaciones pueden realizarse en tiempo logarítmico $O(\log n)$, lo que hace que esta estructura sea especialmente útil para aplicaciones que requieren búsquedas frecuentes y ordenamiento dinámico de datos.

Los árboles son estructuras recursivas, ya que cada subárbol es también un árbol. Una forma particular es el árbol vacío. (Silvia Guardati, 1 enero 2006)

Implementación

Estructura General del Código

El proyecto está organizado en tres archivos principales que implementan la estructura del árbol binario y su interfaz de usuario. La arquitectura sigue un diseño modular que separa las responsabilidades y facilita el mantenimiento del código.

```
1  Estructura del Proyecto
2
3  app/
4  |
5  ├── nodo.py          # Clase base para los nodos del árbol
6  |   └─ class Nodo    # Implementación del nodo básico
7  |
8  ├── arbol_binario.py # Implementación principal del árbol
9  |   └─ class ArbolBinario # Lógica de la estructura de datos
10 |
11 ├── main_sep.py      # Versión con separador de comas
12 |   ├── mostrar_menu() # Función del menú principal
13 |   ├── mostrar_todos_resultados() # Función para mostrar todos los análisis
14 |   └─ main()         # Punto de entrada del programa
15 |
16 └─ main_sep_space.py # Versión con separador de espacios
17     ├── mostrar_menu() # Función del menú principal
18     ├── mostrar_todos_resultados() # Función para mostrar todos los análisis
19     └─ main()         # Punto de entrada del programa
20
```

Ilustración 1, Estructura de la app

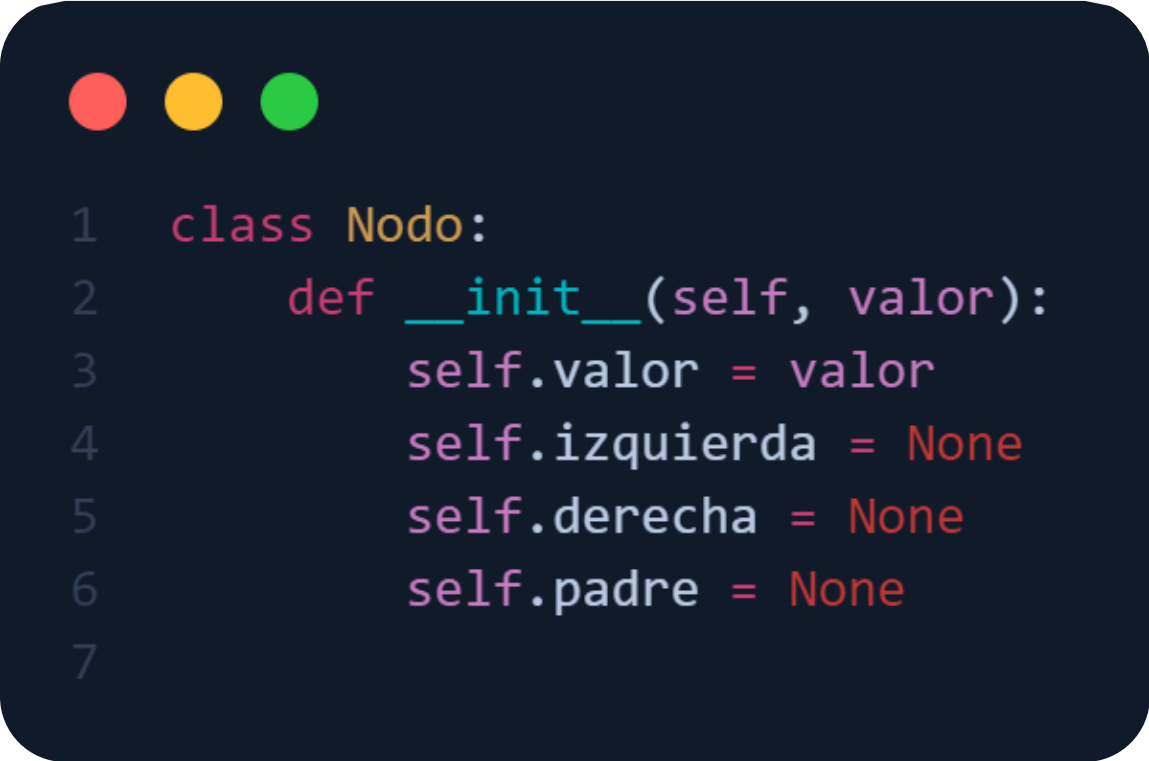
Descripción de Componentes

4.2.1. Clase Nodo

La clase Nodo representa la unidad básica de almacenamiento en el árbol binario. Cada nodo contiene:

- Un valor numérico
- Referencias a sus hijos (izquierdo y derecho)
- Referencia a su nodo padre
- Métodos para gestionar las conexiones entre nodos

Código base de la clase Nodo:



```
1 class Nodo:
2     def __init__(self, valor):
3         self.valor = valor
4         self.izquierda = None
5         self.derecha = None
6         self.padre = None
7
```

Ilustración 2, Estructura de la clase Nodo archivo nodo.py

Clase Árbol Binario

La clase ArbolBinario implementa la lógica principal de la estructura de datos y contiene:

- Método de inserción:

```
1 class ArbolBinario:
2     def __init__(self):
3         # Inicializa el árbol binario con la raíz como None
4         self.raiz = None
5
6     def insertar(self, valor):
7         from nodo import Nodo
8         # Crea un nuevo nodo con el valor dado
9         nuevo_nodo = Nodo(valor)
10
11        # Si la raíz es None, el nuevo nodo se convierte en la raíz
12        if not self.raiz:
13            self.raiz = nuevo_nodo
14            return
15
16        actual = self.raiz
17        while True:
18            # Si el valor es menor que el valor del nodo actual, va a la izquierda
19            if valor < actual.valor:
20                if actual.izquierda is None:
21                    actual.izquierda = nuevo_nodo
22                    nuevo_nodo.padre = actual
23                    break
24                actual = actual.izquierda
25            else:
26                # Si el valor es mayor o igual que el valor del nodo actual, va a la derecha
27                if actual.derecha is None:
28                    actual.derecha = nuevo_nodo
29                    nuevo_nodo.padre = actual
30                    break
31                actual = actual.derecha
```

Ilustración 3, Clase, inicialización y método insertar del archivo arbol_binario.py

Lógica implementada:

```

1  REGLAS DE INSERCIÓN
2  -----
3  - Si el valor es menor que el nodo actual → va a la izquierda
4  - Si el valor es mayor o igual que el nodo actual → va a la derecha
5  - Si no hay nodo en la posición correspondiente → se inserta el nuevo nodo
6  - Si hay nodo → se continúa comparando con ese nodo
7
8  6. EJEMPLO DE ÁRBOL RESULTANTE
9  -----
10         18
11        /  \
12       5    20
13      / \  / \
14     3  8 19 21
15    / \      \
16   1  4        22

```

Ilustración 4, Lógica aplicada a la inserción de los datos(nodos)

```
1  Entrada de datos: 10
2  18 5 3 4 1 8 20 19 21 22
3  Inserción de 18:
4  Primer valor, se convierte en la raíz
5      18
6  Inserción de 5:
7  5 < 18, va a la izquierda
8      18
9      /
10     5
11 Inserción de 3:
12 3 < 18, va a la izquierda
13 3 < 5, va a la izquierda de 5
14     18
15     /
16     5
17     /
18     3
19 Inserción de 4:
20 4 < 18, va a la izquierda
21 4 < 5, va a la izquierda
22 4 > 3, va a la derecha de 3
23     18
24     /
25     5
26     /
27     3
28     \
29     4
```

Ilustración 5, Flujo de la lógica

- Recorrido inorden:

```
1 def recorrido_inorden(self):
2     resultado = []
3
4     def inorden(nodo):
5         # Realiza un recorrido inorden (izquierda, raíz, derecha)
6         if nodo:
7             inorden(nodo.izquierda)
8             resultado.append(nodo.valor)
9             inorden(nodo.derecha)
10
11     inorden(self.raiz)
12     return resultado
```

Ilustración 6, método recorrido_inorden de la clase ArbolBinario

- Funciones de análisis de nodos con 2 hijos:

```
1 def nodos_dos_hijos(self):
2     resultado = []
3
4     def verificar(nodo):
5         # Verifica si un nodo tiene dos hijos
6         if nodo:
7             verificar(nodo.izquierda)
8             if nodo.izquierda and nodo.derecha:
9                 resultado.append(nodo.valor)
10            verificar(nodo.derecha)
11
12     verificar(self.raiz)
13     return resultado
```

Ilustración 7, método nodos_dos_hijos de la clase ArbolBinario

- Búsqueda de un nodo con al menos un hijo par:

```
1 def nodos_hijo_par(self):
2     resultado = []
3
4     def verificar(nodo):
5         # Verifica si un nodo tiene al menos un hijo con valor par
6         if nodo:
7             if ((nodo.izquierda and nodo.izquierda.valor % 2 == 0) or
8                 (nodo.derecha and nodo.derecha.valor % 2 == 0)):
9                 resultado.append(nodo.valor)
10            verificar(nodo.izquierda)
11            verificar(nodo.derecha)
12
13     verificar(self.raiz)
14     return resultado
```

Ilustración 8, método nodos_hijo_par de la clase ArbolBinario

- Suma de los hijos:

```
1 def suma_hijos(self):
2     resultado = []
3
4     def calcular(nodo):
5         # Calcula la suma de los valores de los hijos de cada nodo
6         if nodo:
7             suma = 0
8             suma += nodo.izquierda.valor if nodo.izquierda else 0
9             suma += nodo.derecha.valor if nodo.derecha else 0
10            resultado.append(suma)
11            calcular(nodo.izquierda)
12            calcular(nodo.derecha)
13
14     calcular(self.raiz)
15     return resultado
```

Ilustración 9, método suma_hijos de la clase ArbolBinario

- Búsqueda de caminos

```
1 def encontrar_camino(self, valor):
2     def buscar(nodo):
3         # Busca un nodo con el valor dado
4         if not nodo:
5             return None
6         if nodo.valor == valor:
7             return nodo
8         if valor < nodo.valor:
9             return buscar(nodo.izquierda)
10        return buscar(nodo.derecha)
11
12    nodo = buscar(self.raiz)
13    if not nodo:
14        return None
15
16    camino = []
17    # Encuentra el camino desde la raíz hasta el nodo con el valor dado
18    while nodo:
19        camino.insert(0, nodo.valor)
20        nodo = nodo.padre
21    return camino
```

Ilustración 10, método encontrar_camino de la clase ArbolBinario

Archivo main sep,.py

Principales operaciones implementadas:

- insertar(): Añade nuevos elementos manteniendo el orden
- recorrido_inorden(): Obtiene elementos en orden ascendente
- nodos_dos_hijos(): Identifica nodos con dos descendientes
- nodos_hijo_par(): Encuentra nodos con hijos de valor par
- suma_hijos(): Calcula la suma de los valores de los hijos
- encontrar_camino(): Determina la ruta hacia un nodo específico

4.2.3. Interfaz de Usuario

La interfaz de consola proporciona:

- Menú interactivo con 7 opciones
- Visualización clara de resultados
- Manejo de errores de entrada
- Formato de salida personalizado


```

1 from arbol_binario import ArbolBinario
2
3 def mostrar_menu():
4     print("\n** MENÚ DE OPERACIONES **")
5     print("1. Mostrar recorrido en orden")
6     print("2. Mostrar nodos con dos hijos")
7     print("3. Mostrar nodos con hijo par")
8     print("4. Mostrar suma de hijos")
9     print("5. Buscar camino hacia un nodo")
10    print("6. Mostrar todos los resultados")
11    print("7. Salir")
12    return input("Seleccione una opción: ")
13
14 def mostrar_todos_resultados(arbol):
15    print("\nRecorrido en orden:")
16    print(" ".join(map(str, arbol.recorrido_inorden())))
17
18    print("\nLos que tienen 2 hijos:")
19    print(" ".join(map(str, arbol.nodos_dos_hijos())))
20
21    print("\nLos nodos que tienen 1 hijo par:")
22    print(" ".join(map(str, arbol.nodos_hijo_par())))
23
24    print("\nSuma de sus hijos:")
25    print(" ".join(map(str, arbol.suma_hijos())))
26
27    camino = arbol.encontrar_camino(4) # Valor por defecto: 4
28    print("\nBuscando el camino hacia el nodo 4:")
29    if camino:
30        print(f"El camino es: {' '.join(map(str, camino))}")
31    else:
32        print("El nodo no existe")
33
34 def main():
35    arbol = ArbolBinario()
36
37    # Datos estáticos sacados del modelo de el documento de la actividad
38    valores = [18, 5, 3, 4, 1, 8, 20, 19, 21, 22]
39
40    # Construcción del árbol
41    for valor in valores:
42        arbol.insertar(valor)
43
44    while True:
45        opcion = mostrar_menu()
46
47        if opcion == "1":
48            print("\nRecorrido en orden:")
49            print(" ".join(map(str, arbol.recorrido_inorden())))
50
51        elif opcion == "2":
52            print("\nLos que tienen 2 hijos:")
53            print(" ".join(map(str, arbol.nodos_dos_hijos())))
54
55        elif opcion == "3":
56            print("\nLos nodos que tienen 1 hijo par:")
57            print(" ".join(map(str, arbol.nodos_hijo_par())))
58
59        elif opcion == "4":
60            print("\nSuma de sus hijos:")
61            print(" ".join(map(str, arbol.suma_hijos())))
62
63        elif opcion == "5":
64            try:
65                nodo_buscar = int(input("\nIngrese el nodo a buscar: "))
66                camino = arbol.encontrar_camino(nodo_buscar)
67                if camino:
68                    print(f"El camino es: {' '.join(map(str, camino))}")
69                else:
70                    print("El nodo no existe")
71            except ValueError:
72                print("Por favor, ingrese un número válido")
73
74        elif opcion == "6":
75            mostrar_todos_resultados(arbol)
76
77        elif opcion == "7":
78            print("\n¡Gracias por usar el programa!")
79            break
80
81        else:
82            print("\nOpción no válida. Por favor, intente de nuevo.")
83
84    if __name__ == "__main__":
85        main()

```

Ilustración 11, Estructura del menú archivo main sep Space.py

Características de la interfaz:

- Opciones numeradas del 1 al 7
- Validación de entradas
- Presentación formateada de resultados
- Opción para mostrar todos los análisis

Estructura general del código

Aplicación principal – Por Espacios (APP):

Permite al usuario realizar consultas:

- Recorrido en orden
- Nodos con dos hijos
- Nodos con hijos pares
- Suma de los hijos
- Muestra todos los resultados a la vez (si el usuario elige la opción 6)
- Funciona en bucle hasta que el usuario elige “Salir”

Salida:

```
Recorrido en orden:
1 3 4 5 8 18 19 20 21 22

Los que tienen 2 hijos:
3 5 18 20

Los nodos que tienen 1 hijo par:
18 5 3 21

Suma de sus hijos:
25 11 5 0 0 0 40 0 22 0

Buscando el camino hacia el nodo 4:
El camino es: 18 5 3 4
```

Ilustración 12, Salida del archivo “main sep Space.py” con la opción 6

Aplicación principal – Por Comas (APP):

Permite al usuario realizar consultas:

- Recorrido en orden
- Nodos con dos hijos
- Nodos con hijos pares
- Suma de los hijos
- Muestra todos los resultados a la vez (si el usuario elige la opción 6)
- Funciona en bucle hasta que el usuario elige

Salida:

```
Recorrido en orden:
1 3 4 5 8 18 19 20 21 22

Los que tienen 2 hijos:
3 5 18 20

Los nodos que tienen 1 hijo par:
18 5 3 21

Suma de sus hijos:
25 11 5 0 0 0 40 0 22 0

Buscando el camino hacia el nodo 4:
El camino es: 18 5 3 4
```

Ilustración 13, Salida del archivo “main sep, .py” con la opción 6

Estructura del programa

Video explicativo

Para la comprensión del programa se ha realizado un video explicativo, donde se muestra desde la estructura del programa hasta la ejecución de este. El cual fue subido a la plataforma YouTube y podemos verlo en el siguiente enlace: <https://youtu.be/00pc4j97oQc>

Como ejecutarlo

Para la ejecución del programa debemos ejecutar el código en el archivo “*main sep Space.py*” o “*main sep, .py*” la diferencia de estos dos solo es en la presentación de la salida pero obtendremos los mismos resultados este una vez ejecutado iniciará la vista menú previa configurada.

```
** MENÚ DE OPERACIONES **  
1. Mostrar recorrido en orden  
2. Mostrar nodos con dos hijos  
3. Mostrar nodos con hijo par  
4. Mostrar suma de hijos  
5. Buscar camino hacia un nodo  
6. Mostrar todos los resultados  
7. Salir  
Seleccione una opción:
```

Figura 11, Salida al ejecutar el main.py del programa Python

Salidas

Las salidas que puede generarse al ejecutar el flujo son muy diversas ya que la dinámica es algo amplia, a continuación, se mostrara un resultado.

```
Recorrido en orden:  
[1, 3, 4, 5, 8, 18, 19, 20, 21, 22]  
  
Los que tienen 2 hijos:  
[3, 5, 18, 20]  
  
Los nodos que tienen 1 hijo par:  
[18, 5, 3, 21]  
  
Suma de sus hijos:  
[25, 11, 5, 0, 0, 0, 40, 0, 22, 0]  
  
Buscando el camino hacia el nodo 4:  
El camino es: [18, 5, 3, 4]
```

Figura 12, Resultado de la ejecución del código al ejecutar con la Opción 6

Conclusiones

El desarrollo de este eje permitió comprender la importancia y utilidad de las estructuras de datos tipo Árbol en el manejo eficiente de información. A través de la implementación en Python, se demostró cómo esta estructura permite optimizar búsquedas y almacenamiento de datos, ofreciendo una organización clara y accesible.

Asimismo, se evidenció que los Árboles pueden adaptarse a diversas necesidades, desde la búsqueda de información hasta la optimización de bases de datos. La programación orientada a objetos fue clave en la implementación, facilitando la modularidad y reutilización del código. Finalmente, el programa desarrollado brinda una herramienta interactiva que permite al usuario explorar de manera práctica las funcionalidades de los Árboles en la computación.

Referencias

Meneses, E. (26 de 11 de 2021). *Clase8-Arboles.pdf*. Obtenido de ESTRUCTURAS DE

DATOS: <https://www.uv.mx/personal/ermeneses/files/2021/08/Clase8-Arboles.pdf>

Silvia Guardati, O. (1 enero 2006). *Estructuras de Datos - 3ra Edición*. McGraw Hill Education.