

Árboles en Python

Alumnos:

Ezequiel Franco Sanabria -

ezequiel.sanabria1@gmail.com

Gonzalo Tozzi - tozzigonzalo@gmail.com

Materia:

Programación I

Profesor/a:

Bruselario, Sebastian

Fecha de Entrega: 09/06/2025

Índice

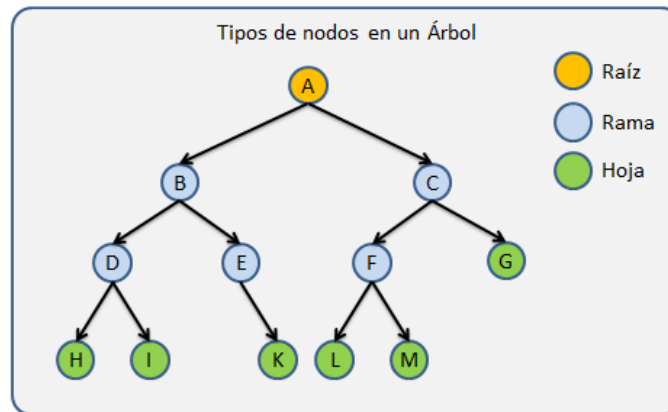
1. Introducción
2. Marco Teórico
3. Caso Práctico
4. Metodología Utilizada
5. Conclusión y Reflexión
6. Bibliografía
7. Anexo

1. Introducción.

En programación, es esencial contar con herramientas que permitan manejar la información de forma ordenada y eficiente, y para eso se utilizan las estructuras de datos. Existen diferentes tipos, como listas, pilas, colas y árboles, cada una con usos y ventajas particulares. Entre ellas, los árboles son estructuras jerárquicas que representan relaciones entre elementos de forma clara y ordenada. Dentro de esta categoría, los árboles binarios son especialmente importantes, ya que cada nodo puede tener como máximo dos hijos, lo que hace que sean ideales para realizar búsquedas, ordenar datos o representar expresiones. Por estas razones, se eligió investigar los árboles binarios en Python, con el objetivo de comprender su funcionamiento y su aplicación práctica en el desarrollo de programas.

2. Marco Teórico.

Un árbol es una estructura de datos jerárquica que se utiliza para organizar información de forma que cada elemento esté conectado a uno o más elementos llamados hijos, formando una especie de árbol.



En nuestro caso vamos a hablar específicamente de una forma de árboles, los árboles binarios. Se caracterizan porque cada nodo puede tener como máximo dos hijos. Estos son muy utilizados en la organización de datos o en los algoritmos de búsqueda y ordenamiento.

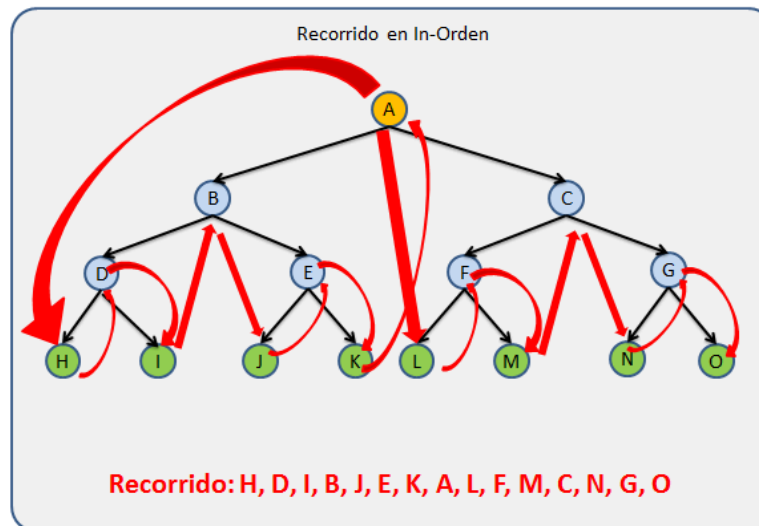
Algunas de los beneficios que tiene esta estructura en particular son:

- Búsqueda eficiente: Encontrar un valor puede hacerse rápidamente, reduciendo el tiempo de búsqueda en comparación con las listas simples.

- Inserción y eliminación ordenada: Permite agregar o quitar elementos manteniendo la estructura.

- Versatilidad en recorridos: Dependiendo de la forma en la que se recorra el árbol (como puede ser inorden o preorden) se puede procesar la información según la necesidad del usuario.

Nosotros utilizamos la forma de recorrer llamada inorden, que hace que los resultados aparecen ordenados en forma ascendente.



3. Caso Práctico

En este trabajo práctico se decidió hacer una **agenda de contactos usando un árbol binario**. Elegimos esta estructura porque permite guardar y buscar los contactos de forma rápida y ordenada. Cada contacto será un nodo del árbol, y así se pueden agregar, buscar o eliminar fácilmente. Es una forma práctica de aplicar lo aprendido sobre árboles binarios.

4. Metodología Utilizada

- Instalación de Visual Studio Code:
 - El lenguaje de programación utilizado fue Python 3.11.9.
- Se investigó con el material del aula y fuentes externas acerca de los árboles.
 - Se tomó nota de los conceptos que más responden a los Árboles Binarios, especialmente sobre Árboles Binarios de Búsqueda
- Se interiorizó en el tema de los árboles binarios.
 - Se hizo una redacción del marco teórico
- La búsqueda de opciones para realizar el caso práctico.
 - La decisión fue hacer una agenda de contactos telefónicos.

- Una reunión virtual para que se decida cómo iba a ser el programa, y para repartir las funciones.
 - Mediante GitHub se ha realizado el trabajo en conjunto y la utilización de comandos como: commit, branch, push y pull.
- Se hicieron pruebas para corroborar que el programa funcione correctamente.
 - Las pruebas fueron grabadas con herramientas de captura de video de Zoom

5. Conclusión

Lo que se pudo notar de los árboles binarios de búsqueda es que son una estructura de datos útil cuando se requiere insertar, buscar y eliminar estos de manera eficiente. Pero no siempre es la mejor opción.

Se puede ver que son muy eficientes cuando se necesita acceder a los datos ordenados gracias a los recorridos en orden que permite encontrar los valores mínimos o máximos rápidamente, o cuando hay muchas inserciones y eliminaciones de datos, estos árboles se pueden modificar sin tener que reordenar todos los datos.

Pero también hay casos donde no son eficientes. Un ejemplo es cuando no está balanceado el árbol (por ejemplo, si se insertan los datos ya ordenados) esto haría que la búsqueda sea igual a una lista ordenada.

Otro caso donde no es tan eficiente es cuando el número de elementos es pequeño, para estos casos es mejor alguna estructura más simple como puede ser una lista.

6. Bibliografía

Takeyas, B. (s.f.). *Árboles binarios de búsqueda (ABB)*. Instituto Tecnológico Nacional de México (TecNM), Campus Naranjos.

<https://nlaredo.tecnm.mx/takeyas/Apuntes/Estructura%20de%20Datos/Apuntes/07-ABB.pdf>

Universidad Tecnológica Nacional (UTN), Facultad Regional San Nicolás. (s.f.). *Árboles binarios de búsqueda*. Google Docs.

https://docs.google.com/document/d/10k16oL15EeyOaq92aoi4qwK3t_22X29-FSV2iV-8N1U/edit

Valdés, A. G. (s.f.). *Binary Search Trees*. Universidad Técnica Federico Santa María.

<http://profesores.elo.utfsm.cl/~agv/elo320/01and02/dataStructures/binarySearchTree.pdf>

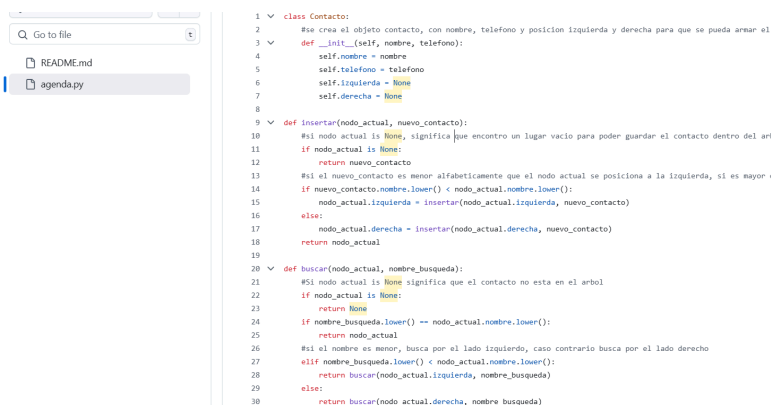
Vega, J. F. (s.f.). *Árboles binarios de búsqueda*. Universidad de Granada.

https://ccia.ugr.es/~jfv/ed1/tedi/cdrom/docs/arb_BB.htm

7. Anexo

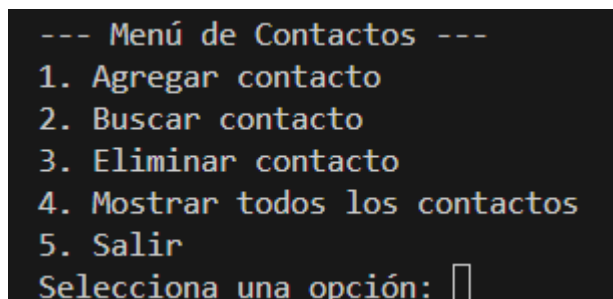
Enlace al repositorio del proyecto: https://github.com/GonzaloTozzi/TP_Int_2

Captura de GitHub:



```
1 class Contacto:
2     #Se crea el objeto contacto, con nombre, telefono y posicion izquierda y derecha para que se pueda armar el
3     def __init__(self, nombre, telefono):
4         self.nombre = nombre
5         self.telefono = telefono
6         self.izquierda = None
7         self.derecha = None
8
9     def insertar(nodo_actual, nuevo_contacto):
10        #Si nodo actual es None, significa que encontro un lugar vacio para poder guardar el contacto dentro del arb
11        if nodo_actual is None:
12            return nuevo_contacto
13        #Si el nuevo_contacto es menor alfabeticamente que el nodo actual se posiciona a la izquierda, si es mayor o
14        if nuevo_contacto.nombre.lower() < nodo_actual.nombre.lower():
15            nodo_actual.izquierda = insertar(nodo_actual.izquierda, nuevo_contacto)
16        else:
17            nodo_actual.derecha = insertar(nodo_actual.derecha, nuevo_contacto)
18        return nodo_actual
19
20    def buscar(nodo_actual, nombre_busqueda):
21        #Si nodo actual es None significa que el contacto no esta en el arbol
22        if nodo_actual is None:
23            return None
24        if nombre_busqueda.lower() == nodo_actual.nombre.lower():
25            return nodo_actual
26        #Si el nombre es menor, busca por el lado izquierdo, caso contrario busca por el lado derecho
27        elif nombre_busqueda.lower() < nodo_actual.nombre.lower():
28            return buscar(nodo_actual.izquierda, nombre_busqueda)
29        else:
30            return buscar(nodo_actual.derecha, nombre_busqueda)
```

Capturas del programa:



```
--- Menú de Contactos ---
1. Agregar contacto
2. Buscar contacto
3. Eliminar contacto
4. Mostrar todos los contactos
5. Salir
Selecciona una opción: █
```

```
Selecciona una opción: 1
Nombre: Ezequiel
Teléfono: 12345

Contacto agregado.
```

```
Selecciona una opción: 2
Nombre del contacto a buscar: ezequiel
Contacto encontrado: Ezequiel, Teléfono: 12345
```

```
Selecciona una opción: 3
Nombre del contacto a eliminar: ezequiel

Contacto eliminado.
```

```
Selecciona una opción: 2
Nombre del contacto a buscar: ezequiel

Contacto no encontrado.
```