UTN

Programación 1

Trabajo Práctico Integrador: Árboles Binarios

Profesora: Julieta Trapé

Tutor: Sofía Raia

Grupo:

Rodriguez Karina

Ovelar Isaias Javier

Fecha 08/06/25

Índice

1. Introducción

En el amplio universo de la programación, la **gestión y organización de datos** es un pilar fundamental para el desarrollo de software eficiente y robusto. La elección de la estructura de datos correcta no solo optimiza el rendimiento de una aplicación, sino que también minimiza los costos asociados a los recursos de hardware necesarios para su ejecución, un aspecto crucial en el mundo real donde la eficiencia se traduce directamente en ahorro económico.

Dentro de las diversas estructuras de datos disponibles, los **árboles** se destacan por su versatilidad y amplia aplicabilidad. Su naturaleza jerárquica los convierte en la solución ideal para modelar una gran variedad de escenarios, desde la organización de sistemas de archivos y directorios en un ordenador hasta la implementación de complejos **algoritmos de búsqueda y ordenamiento de datos**. En este trabajo práctico, presentaremos un ejemplo de uso de árboles, particularmente de árboles binarios.

2. Marco Teórico

Un **árbol binario** es una **estructura de datos no lineal** y jerárquica que organiza la información de una manera particular, asemejándose visualmente a la estructura ramificada de un árbol invertido. Su característica principal es que cada elemento, conocido como **nodo**, puede tener como máximo dos "hijos" directos: un hijo izquierdo y un hijo derecho. Esta

limitación binaria es fundamental para la eficiencia de ciertas operaciones sobre la estructura.

Los componentes principales de un árbol binario son:

- **Raíz**: Es el **nodo superior** y fundamental del árbol. No tiene ningún nodo "padre" y es el punto de entrada para acceder a todos los demás elementos de la estructura. Es la base desde la cual se construye y se recorre todo el árbol.

- **Nodo**: Representa la **unidad mínima** de un árbol binario. Cada nodo, además de contener un **dato** (la información que almacena, como un apellido, un nombre, un DNI, etc.), posee dos referencias:

    - **Referencia izquierda**: Apunta al nodo hijo izquierdo.
    - **Referencia derecha**: Apunta al nodo hijo derecho. En Python, estas referencias se gestionan mediante la asignación de objetos, creando un vínculo directo entre los nodos. En lenguajes de programación de bajo nivel como C o C++, este vínculo se establece explícitamente a través del uso de **punteros**.
- **Nodos Padre e Hijo**: Dentro de la jerarquía de un árbol, cada nodo, a excepción de la raíz, es **hijo** de un nodo de nivel superior, al que se denomina **padre**. Un nodo padre puede tener uno o dos nodos hijos.

- **Nodos Hoja**: Son aquellos nodos que **no poseen ninguna referencia a otros nodos**, es decir, no tienen hijos. Representan los "extremos" de las ramas del árbol.

- **Subárboles**: Cada nodo en un árbol binario puede ser considerado la raíz de un **subárbol**. Un subárbol es, a su vez, un árbol binario que contiene al nodo actual y a todos sus descendientes.

La naturaleza estructurada y las reglas de conexión entre sus nodos hacen de los árboles binarios una herramienta excepcionalmente útil para la organización, búsqueda y recuperación eficiente de datos en diversas aplicaciones informáticas.

Los árboles presentan diversas ventajas que los hacen interesantes para su implementación:

Búsqueda eficiente: En un árbol binario de búsqueda (BST) bien balanceado, la búsqueda de un elemento toma un tiempo promedio de O(logn), donde n es el número de nodos. Esto es significativamente más rápido que la búsqueda lineal en una lista o array no ordenado, que toma O(n) tiempo. Esta eficiencia se debe a que en cada paso de la búsqueda, el algoritmo descarta la mitad de los elementos restantes.

Inserción y eliminación rápidas: Al igual que la búsqueda, las operaciones de inserción y eliminación en un BST balanceado también tienen una complejidad temporal promedio de O(logn). Esto permite añadir o quitar elementos de la estructura de datos de forma eficiente, sin necesidad de reorganizar una gran cantidad de datos como ocurriría en un array.

Organización jerárquica y relaciones claras: Los árboles binarios representan naturalmente relaciones jerárquicas entre los datos, lo que los hace ideales para modelar ciertas estructuras del mundo real, como sistemas de archivos, árboles genealógicos o la estructura de un documento HTML (DOM). La relación padre-hijo es intrínseca a su diseño.

Flexibilidad en la estructura de datos: A diferencia de los arrays, que tienen un tamaño fijo o requieren reasignación para crecer, los árboles binarios pueden crecer o encogerse dinámicamente a medida que se añaden o eliminan elementos. Esto los hace muy adaptables a situaciones donde la cantidad de datos puede variar.

Base para estructuras más complejas: Los árboles binarios son la base de muchas otras estructuras de datos más avanzadas y especializadas, como los **árboles binarios de búsqueda auto-balanceados** (AVL, Rojo-Negro), que garantizan la eficiencia logarítmica incluso en el peor de los casos, o los **montículos (heaps)**, utilizados en algoritmos de ordenamiento y colas de prioridad.

Recorridos eficientes: Los árboles binarios permiten diferentes tipos de recorridos (inorden, preorden, postorden) que facilitan el procesamiento de los datos de diversas maneras.

A la hora de trabajar con árboles binarios, la **recursividad** se convierte en una técnica sumamente práctica y elegante para implementar sus operaciones fundamentales. Su naturaleza auto-referencial se alinea perfectamente con la estructura intrínseca del árbol.

Para **agregar nodos** al árbol, el proceso generalmente sigue estos pasos:

1. Primero, se evalúa si la **raíz** del árbol está vacía. Si es así, el nuevo nodo se establece directamente como la nueva raíz.
2. Si la raíz ya contiene un nodo, se compara el valor del dato del nuevo nodo con el dato del nodo actual.
    - Si el nuevo dato es **menor**, el algoritmo se desplaza hacia el **subárbol izquierdo**.
    - Si el nuevo dato es **mayor**, el algoritmo se desplaza hacia el **subárbol derecho**.
3. Este proceso de comparación y desplazamiento se repite recursivamente hasta que se encuentra un nodo vacío (una posición donde el hijo izquierdo o derecho es null/None), momento en el cual el nuevo nodo se inserta en esa ubicación.

Para **recorrer el árbol** y acceder a sus elementos de manera sistemática, existen tres métodos principales, todos ellos implementados de forma natural con recursividad:

- **Recorrido Preorden (PreOrder)**:

1. Se procesa el **nodo actual** (se muestra su contenido).
2. Se visita recursivamente el **subárbol izquierdo**.
3. Se visita recursivamente el **subárbol derecho**. Este recorrido es útil para crear una copia del árbol o para evaluar expresiones.

- **Recorrido Inorden (InOrder)**:

1. Se visita recursivamente el **subárbol izquierdo**.
2. Se procesa el **nodo actual** (se muestra su contenido).
3. Se visita recursivamente el **subárbol derecho**. En un **Árbol Binario de Búsqueda (BST)**, un recorrido inorden siempre produce los elementos ordenados de forma ascendente.

- **Recorrido Postorden (PostOrder)**:

1. Se visita recursivamente el **subárbol izquierdo**.
2. Se visita recursivamente el **subárbol derecho**.
3. Se procesa el **nodo actual** (se muestra su contenido). Este recorrido es comúnmente utilizado para eliminar nodos de un árbol de forma segura, ya que asegura que los hijos se procesen antes que el padre.

3. Caso Práctico

Un desafío constante en el ámbito de la informática es la **organización y el mantenimiento ordenado de grandes volúmenes de datos**. Ante esta necesidad, las **estructuras de datos tipo árbol** emergen como una solución excepcionalmente eficiente y versátil.

En el presente proyecto, nos propusimos demostrar de manera práctica cómo los **árboles binarios de búsqueda (BST)** permiten organizar la información. Para ello, hemos desarrollado un sistema que gestiona **datos de personas**, utilizando el apellido como clave principal de ordenamiento.

Nuestro caso práctico se centra en la implementación de un **árbol binario de apellidos**, donde cada nodo del árbol no solo almacena un apellido, sino que también contiene una **lista de objetos Persona**. Esta lista es crucial, ya que permite agrupar a todas las personas que comparten el mismo apellido en un único nodo del árbol. De esta forma, el sistema puede manejar de manera eficiente situaciones donde múltiples individuos poseen el mismo apellido, manteniendo la integridad y coherencia de los datos.

A través de esta implementación, se mostrará cómo se realizan operaciones fundamentales como:

- **Inserción de nuevas personas**: Agregando sus datos y ubicándose correctamente en el árbol según su apellido.
- **Búsqueda de individuos**: Localizando rápidamente a una persona específica o a todas las personas con un apellido determinado.
- **Modificación y eliminación de registros**: Permitiendo la actualización o el retiro de datos de personas del sistema.

Este enfoque práctico ilustra la capacidad de los árboles binarios para ofrecer una **organización jerárquica y un acceso optimizado** a la información, demostrando su relevancia en escenarios reales de gestión de datos.

4. Metodología Utilizada

En este proyecto, hemos desarrollado un sistema que ilustra el funcionamiento básico de un **árbol binario de búsqueda** (**BST**) combinado con **listas**. Nuestro objetivo es gestionar eficientemente la información de personas, organizándose por apellido y permitiendo el acceso rápido a sus datos.

El sistema se compone de 3 clases fundamentales, y 2 módulos:

Clase Persona

Esta clase define la estructura de datos para cada individuo. Contiene los atributos esenciales para representar a una persona:

- **apellido**: La cadena de texto que almacena el apellido de la persona.
- **nombre**: La cadena de texto que almacena el nombre de la persona.
- **DNI**: Un valor numérico o cadena de texto que representa el Documento Nacional de Identidad de la persona.

Además, incluye el **método mágico __str__()**, que proporciona una representación legible y concisa de los atributos de la persona cuando se imprime el objeto.

Clase Nodo

La clase Nodo es la pieza fundamental para construir la estructura del árbol. Cada instancia de Nodo representa un elemento en el árbol y se compone de los siguientes atributos:

- **apellido**: Este atributo es clave para la organización del árbol. Los nodos se ordenan alfabéticamente según este apellido, lo que facilita las operaciones de búsqueda y recorrido.
- **personas**: Una **lista** (o arreglo) diseñada para almacenar múltiples objetos del tipo Persona. Esto permite que un mismo apellido pueda asociarse con varias personas diferentes (ej. "González" puede tener múltiples individuos con ese apellido).
- **izquierda**: Una referencia (puntero) al nodo hijo izquierdo. En un BST, este nodo contendrá apellidos alfabéticamente menores que el apellido del nodo actual.
- **derecha**: Una referencia (puntero) al nodo hijo derecho. Este nodo contiene apellidos alfabéticamente mayores que el apellido del nodo actual.

También incorpora el **método mágico __str__()**, el cual itera sobre la lista personas[] y presenta de forma secuencial el contenido de cada objeto Persona almacenado en el nodo.

Clase ArbolBinario

Esta clase encapsula la lógica principal para operar sobre el árbol binario. Mantiene la **raíz** del árbol y ofrece una serie de métodos para manipular la estructura:

- **raiz**: Es la referencia al nodo superior del árbol. En Python, es el punto de entrada para acceder a todos los demás nodos.

- **__str__()**: Este método mágico permite obtener una representación en cadena de todo el contenido del árbol. Para lograrlo, invoca a la función recursiva **mostrarInordenRecursivo()**, que garantiza un recorrido ordenado de los apellidos.

- **insertarNodo()**: Este método público se encarga de iniciar el proceso de inserción de un nuevo nodo en el árbol. Internamente, delega la lógica recursiva a **insertarNodoRecursivo()**.

- **insertarNodoRecursivo()**: Una función recursiva que recorre el árbol para encontrar la posición adecuada para un nuevo nodo. Si llega a una posición vacía (un nodo hoja), el nuevo nodo se almacena allí. De lo contrario, evalúa el atributo apellido del nuevo nodo: si es alfabéticamente mayor que el apellido del nodo actual, se desplaza hacia la rama derecha; si es menor, se desplaza hacia la rama izquierda. Este proceso continúa hasta encontrar un lugar disponible.

- **buscarNodoRecursivo()**: Un método recursivo diseñado para buscar un nodo específico dentro del árbol basándose en su atributo apellido. Si el nodo es encontrado, devuelve una referencia a dicho nodo; de lo contrario, retorna None.

- **agregarPersona()**: Este método permite añadir un nuevo objeto Persona al arreglo personas[] dentro del nodo correspondiente del árbol. Utiliza buscarNodoRecursivo() para localizar el nodo con el apellido de la persona. Si el nodo existe, la persona se agrega a su lista. Si el nodo no se encuentra, se crea un nuevo nodo utilizando insertarNodo() y, posteriormente, la persona se añade a la lista del recién creado

nodo.

- **mostrarInordenRecursivo()**: Esta función recursiva es fundamental para visualizar el contenido del árbol de manera ordenada. Realiza un **recorrido inorden**, lo que significa que procesa primero la rama izquierda, luego el nodo actual, y finalmente la rama derecha, asegurando que los apellidos se muestren en orden ascendente.

- **borrarPersona()**: Este método se encarga de eliminar un objeto Persona específico del árbol, buscando por nombre y apellido. Primero, intenta localizar el nodo que contiene el apellido. Si el nodo no se encuentra, se muestra un mensaje informativo. Si el nodo es encontrado, se busca el objeto Persona dentro de su lista personas[] que coincida con el nombre y se retira utilizando el método remove() de la lista.

- **buscarPersona()**: Opera de manera similar a borrarPersona(), buscando una persona por nombre y apellido. La diferencia principal es que, en lugar de eliminar el objeto, este método simplemente se limita a mostrar todos los atributos del objeto Persona encontrado.

- **Módulo main**

  Este es el módulo principal del proyecto. Contiene la función **main()**, que sirve como punto de inicio de la ejecución del programa, y se encarga de definir y gestionar el **menú de opciones** para interactuar con el árbol binario.

- **Módulo popularArbol**

  Este es un módulo auxiliar que incluye la función **popularArbol()**. Su propósito es facilitar la demostración del funcionamiento del proyecto, ya que carga un **número predefinido de objetos Persona** en el árbol binario. Esto permite iniciar el programa

con datos ya cargados, listos para ser manipulados y consultados, sin necesidad de ingresarlos manualmente cada vez.

5. Resultados Obtenidos

En su conjunto, el proyecto logró su objetivo principal de demostrar el funcionamiento básico de un árbol binario en un contexto práctico de gestión de datos. Sirvió como una clara ilustración de cómo estas estructuras permiten organizar, buscar y manipular información de manera eficiente, superando las limitaciones de otras estructuras lineales para ciertos escenarios.

En conclusión, los resultados obtenidos validan la elección de un árbol binario para la organización de datos jerárquicos, confirmando su eficiencia y flexibilidad en la gestión de registros de personas.

6. Conclusiones

A partir del análisis del código y la metodología propuesta para el proyecto del árbol binario, podemos extraer varias conclusiones clave sobre su diseño y funcionalidad:

---

1. Aplicación Práctica de Estructuras de Datos Combinadas

El proyecto demuestra una aplicación práctica y efectiva de la combinación de dos estructuras de datos fundamentales: **árboles binarios de búsqueda (BST)** y **listas)**. El BST se utiliza para organizar eficientemente los apellidos, permitiendo búsquedas, inserciones y eliminaciones logarítmicas (O(logn) en el caso promedio). La lista interna en cada nodo maneja la posible colisión de apellidos, almacenando múltiples objetos Persona asociados a un mismo apellido. Esta sinergia optimiza tanto la organización global de los datos como la gestión detallada dentro de cada nodo.

---

2. Eficiencia en la Gestión de Datos de Personas

El diseño propuesto permite una gestión eficiente de la información de personas. La estructura jerárquica del árbol, basada en los apellidos, asegura que operaciones como la búsqueda de una persona por apellido sean rápidas. Una vez encontrado el nodo del apellido, la búsqueda dentro de la lista de personas es lineal (O(k), donde k es el número de personas con ese apellido), pero este k suele ser mucho menor que el total de personas en el sistema, lo que mantiene la eficiencia general.

---

3. Claridad y Modularidad del Diseño

La división del proyecto en clases (Persona, Nodo, ArbolBinario) y módulos (main, popularArbol) refleja un buen diseño orientado a objetos y una adecuada modularidad. Cada clase tiene una responsabilidad clara y bien definida, lo que facilita la comprensión, el mantenimiento y la posible extensión del código. El uso de métodos recursivos para las operaciones del árbol (insertarNodoRecursivo, buscarNodoRecursivo, mostrarInordenRecursivo) simplifica la lógica y la hace más elegante para trabajar con estructuras recursivas como los árboles.

---

4. Potencial de Mejora en Balanceo y Manejo de Errores

Si bien el proyecto cumple su objetivo de mostrar el funcionamiento básico de un árbol binario, es importante notar que un BST estándar puede degenerar en una lista enlazada en el peor de los casos (cuando los datos se insertan en un orden estrictamente ascendente o descendente), llevando a una complejidad de O(n) para las operaciones. Una mejora significativa sería la implementación de otro tipo de arbol, en el que la implantación se encuentre realizada sobre arreglos, que permite exceder el límite de 2 ramas por nodo, otorgando mayor flexibilidad.

---

5. Base Sólida para Proyectos Futuros

Este proyecto sienta una base sólida para explorar conceptos más avanzados de estructuras de datos y algoritmos. Sirve como una excelente introducción a la implementación de árboles binarios, la recursividad y la combinación de diferentes tipos de estructuras de datos para resolver problemas específicos. A partir de aquí, se podrían añadir funcionalidades más complejas, como la eliminación de nodos en el árbol (que es una operación más compleja en BSTs), la serialización/deserialización del árbol o la implementación de interfaces gráficas de usuario.

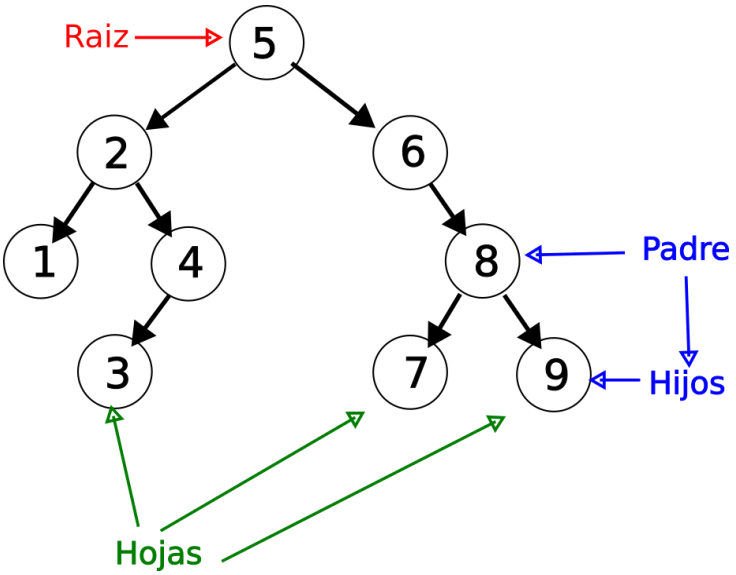
En resumen, el proyecto ofrece una implementación clara y funcional de un sistema de gestión de personas basado en árboles binarios, destacando las ventajas de esta estructura para la organización y recuperación eficiente de datos.

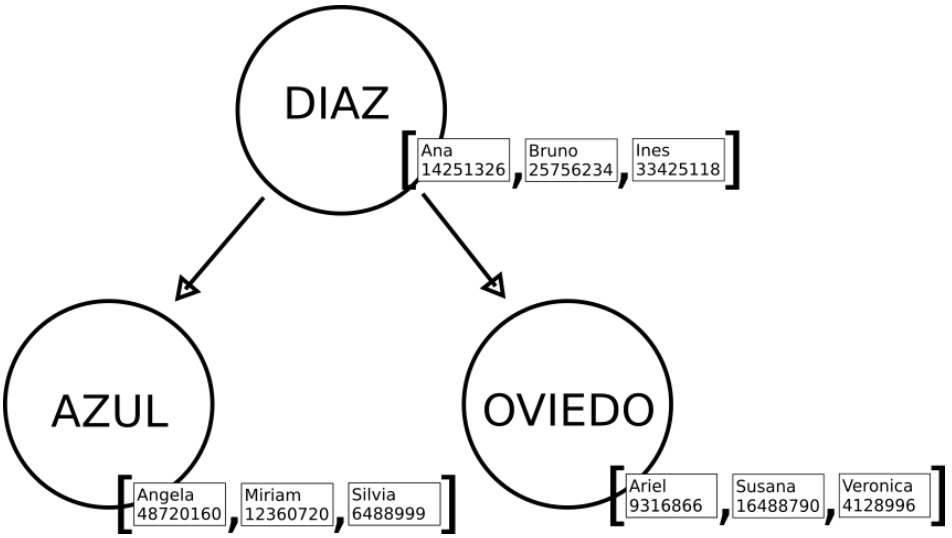
7. Bibliografía

https://medium.com/@matematicasdiscretaslibro/cap%C3%ADtulo-12-teoria-de-arboles-binarios-f731baf470c0

https://es.wikipedia.org/wiki/%C3%81rbol_(inform%C3%A1tica)

8. Anexos






```
------------------------------------------------------------
Bienvenido al arbol de apellidos!
Desea popular el arbol con personas de prueba? (si/no): si

------------------------------------------------------------
1- Agregar una persona
2- Mostrar Arbol de Apellidos y Lista de Personas
3- Borrar una persona
4- Buscar una persona
5- Salir

Ingrese una opcion:
```

```
Ingrese una opcion: 2
-------------------------------------------------------

Arbol completo:


Apellido: 'Gimenez' (1 personas)
- Gimenez Camilo, DNI: 41829534


Apellido: 'Martinez' (1 personas)
- Martinez Sandra, DNI: 32206425


Apellido: 'Ovelar' (1 personas)
- Ovelar Javier, DNI: 28674865


Apellido: 'Perez' (1 personas)
- Perez Eduardo, DNI: 46162544


Apellido: 'Rodriguez' (2 personas)
- Rodriguez Karina, DNI: 40120448
- Rodriguez Sebastian, DNI: 23334574
```

```
-------------------------------------------------------
1- Agregar una persona
2- Mostrar Arbol de Apellidos y Lista de Personas
3- Borrar una persona
4- Buscar una persona
5- Salir

Ingrese una opcion: 3
-------------------------------------------------------

Ingrese el apellido de la persona a eliminar: Antares
No se encuentra el apellido

-------------------------------------------------------
```