**GUÍA DE LABORATORIO NO. 7**

**UNIVERSIDAD MANUELA BELTRÁN**

**ESTRUCTURA DE DATOS**

**DOCENTE**

**HUGO ALFONSO ORTIZ BARRERO**

**BOGOTA DC 05 MAYO 2025**

**SITUACIONES EN LAS QUE SE PRESENTA LA ESTRUCTURA DE ÁRBOL:**

1. **Sistema de archivos de un ordenador**: La jerarquía de carpetas y archivos se organiza en forma de un árbol, con directorios que pueden contener otros directorios y archivos.
2. **Bases de datos jerárquicas**: En bases de datos, como las que usan modelos jerárquicos, los datos se organizan en forma de árbol con nodos padre e hijos.
3. **Árbol genealógico**: La representación de las relaciones familiares entre generaciones, donde cada persona es un nodo y los padres son los nodos padres de sus hijos.
4. **Decisiones en juegos de estrategia**: Los árboles de decisión son utilizados para representar las diferentes opciones posibles y sus consecuencias en los juegos.
5. **Estructuras de decisiones en inteligencia artificial**: Los algoritmos de IA, como los árboles de decisión, utilizan árboles para dividir problemas en subproblemas más pequeños y más manejables.
6. **Representación de expresiones matemáticas**: Las expresiones algebraicas y aritméticas se pueden representar como árboles, donde los nodos son operadores y los operandos son hojas.
7. **Compiladores de lenguajes de programación**: Los compiladores utilizan árboles de sintaxis abstracta para representar la estructura del código fuente.
8. **Redes sociales**: Las conexiones de personas pueden representarse como un árbol, donde cada usuario tiene amigos que son nodos hijos.
9. **Sistemas de navegación**: Los mapas y rutas se pueden organizar en una estructura de árbol donde cada intersección o punto de conexión es un nodo.
10. **Protocolos de comunicación en redes**: Algunos protocolos de comunicación utilizan árboles para organizar y gestionar la transmisión de datos de manera eficiente.

**COMPARATIVA DE ÁRBOLES B, B+ Y AVL:**

|  |  |  |  |
| --- | --- | --- | --- |
| Característica | Árbol B | Árbol B+ | Árbol AVL |
| Definición | Es un árbol balanceado que mantiene los datos en nodos internos, donde cada nodo puede tener múltiples claves e hijos. | Es una variación del árbol B, donde las claves se almacenan solo en los nodos hoja, y las hojas están conectadas en una lista. | Es un árbol binario de búsqueda balanceado en el que la diferencia de alturas entre los subárboles izquierdo y derecho de cualquier nodo es como máximo 1. |
| Balance | Balanceado en términos de número de claves por nodo y altura del árbol. | Similar al árbol B, pero más eficiente en las búsquedas ya que solo se almacenan datos en las hojas. | Balanceado por reglas de rotación que aseguran que el árbol permanezca equilibrado en altura. |
| Estructura de nodos | Los nodos internos y las hojas contienen claves y punteros a otros nodos. | Las claves solo están en los nodos hoja, y los nodos internos solo almacenan punteros. | Cada nodo contiene solo una clave y punteros a nodos hijos. |
| Accesibilidad de los datos | Acceso rápido a los datos, pero las búsquedas pueden involucrar nodos internos y hojas. | Las búsquedas se realizan solo en los nodos hoja, lo que mejora el rendimiento de las búsquedas secuenciales. | Las búsquedas son rápidas, pero las operaciones de inserción y eliminación requieren rotaciones para mantener el balance. |
| Operaciones de inserción y eliminación | Requiere redistribuir las claves entre los nodos para mantener el balance. | Similar al árbol B, pero las inserciones y eliminaciones pueden ser más eficientes debido a la estructura de solo hojas. | Requiere rotaciones para asegurar el balance después de las inserciones o eliminaciones. |
| Velocidad de búsqueda | Eficiente, con tiempo logarítmico en función del número de claves y la altura del árbol. | Muy eficiente en búsquedas secuenciales, ya que las claves están organizadas solo en las hojas. | Rápida, con tiempo logarítmico debido a la naturaleza balanceada del árbol. |
| Uso común | Utilizado en sistemas de bases de datos y sistemas de almacenamiento con grandes cantidades de datos. | Comúnmente usado en sistemas de bases de datos que requieren búsquedas rápidas y rangos de datos. | Usado principalmente en estructuras de datos en memoria donde las inserciones y eliminaciones son frecuentes. |
| Complejidad en implementación | Moderadamente compleja, debido a la necesidad de mantener la estructura balanceada durante las inserciones y eliminaciones. | Más sencilla que el árbol B debido a la estructura de solo hojas y la lista vinculada entre ellas. | Relativamente compleja debido a las rotaciones requeridas para mantener el balance. |
| Espacio adicional | Necesita espacio adicional para almacenar punteros a nodos hijos. | Necesita espacio adicional para almacenar los punteros de las hojas y las conexiones entre ellas. | Requiere espacio adicional para mantener el balance, pero sin la complejidad de nodos múltiples. |

**ÁRBOL DE CÓDIGOS HUFFMAN: APLICACIÓN Y FUNCIONAMIENTO**

El **algoritmo de Huffman** es un algoritmo de compresión sin pérdida que se utiliza ampliamente en la compresión de datos, como en archivos ZIP, imágenes JPEG y en la transmisión de datos en redes. Su objetivo es reducir el tamaño de los datos al asignar códigos de longitud variable a los símbolos de un conjunto, donde los símbolos más frecuentes tienen códigos más cortos y los menos frecuentes tienen códigos más largos.

**Funcionamiento:**

1. **Frecuencia de símbolos**: Se comienza con una lista de símbolos (letras, caracteres, etc.) y su frecuencia de aparición.
2. **Construcción del árbol**: Se construye un árbol binario donde cada nodo contiene un símbolo y su frecuencia. Los dos nodos con las frecuencias más bajas se combinan en un nuevo nodo. Este proceso se repite hasta que todos los nodos se combinan en un solo árbol.
3. **Asignación de códigos**: A cada símbolo se le asigna un código binario según su posición en el árbol. Los símbolos más frecuentes tienen códigos más cortos, mientras que los símbolos menos frecuentes tienen códigos más largos.
4. **Codificación**: Los símbolos se reemplazan por sus códigos binarios, resultando en una representación más compacta de los datos.

Este proceso de compresión es muy útil en la reducción de tamaño de archivos de texto, imágenes o cualquier tipo de datos donde ciertos símbolos se repiten con mayor frecuencia.

**Ejemplo:**

Supongamos que tenemos la siguiente lista de símbolos y sus frecuencias:

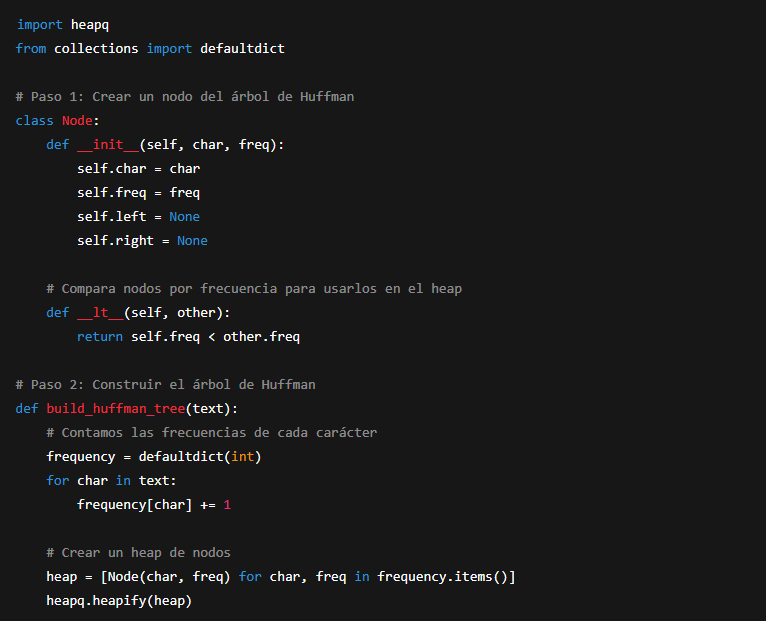
* 'A': 5
* 'B': 9
* 'C': 12
* 'D': 13
* 'E': 16
* 'F': 45

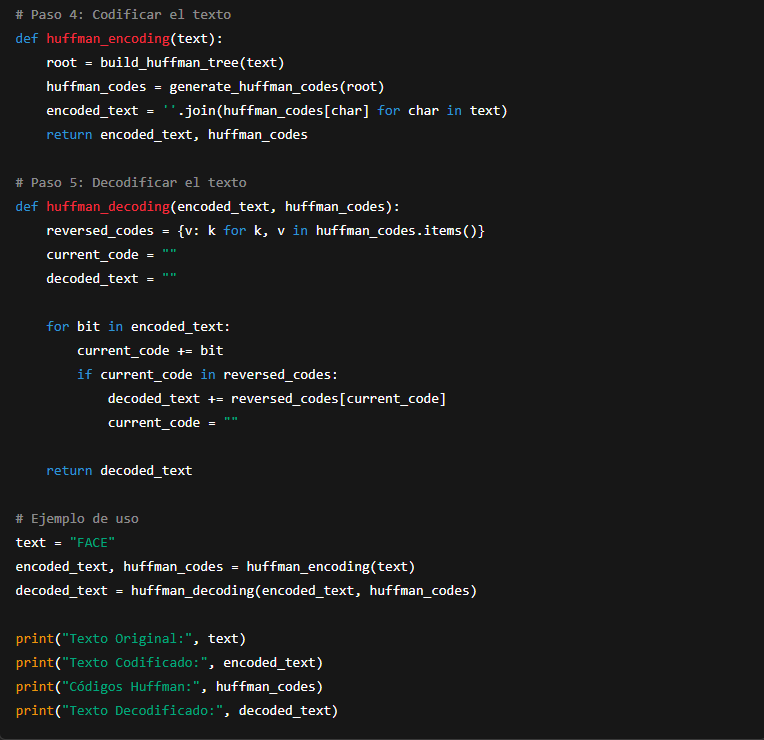
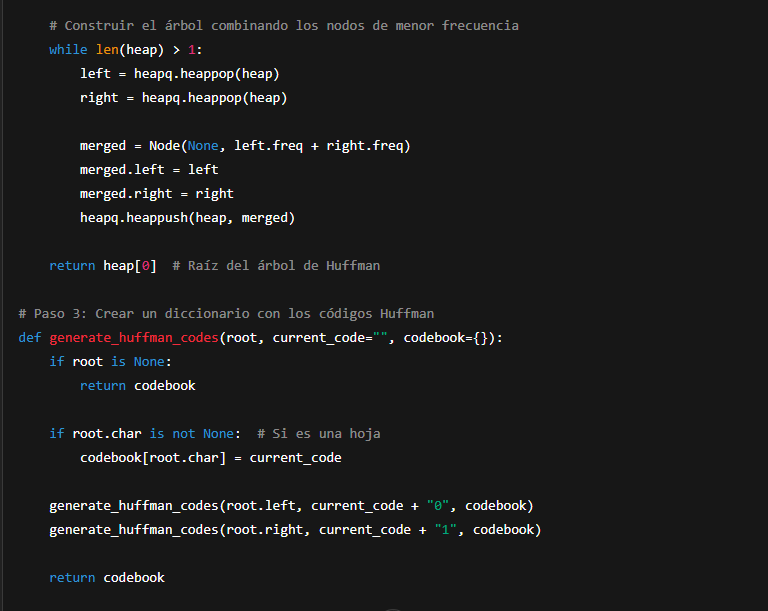
El árbol de Huffman para estos símbolos sería construido de la siguiente manera, combinando los nodos de menor frecuencia hasta formar el árbol final. El código binario asignado a cada símbolo se derivaría del árbol. Los códigos serían algo como:

* 'A': 000
* 'B': 001
* 'C': 01
* 'D': 10
* 'E': 11
* 'F': 1

Por lo tanto, la palabra "FACE" sería codificada como "1 11 000 11".

**Código del algoritmo de Huffman en Python:**





**Explicación del código**:

1. **Node**: Representa los nodos del árbol de Huffman. Cada nodo tiene un carácter (si es una hoja) y una frecuencia.
2. **build\_huffman\_tree**: Construye el árbol de Huffman a partir de las frecuencias de los caracteres en el texto.
3. **generate\_huffman\_codes**: Recorre el árbol de Huffman y asigna códigos binarios a cada carácter.
4. **huffman\_encoding**: Codifica el texto utilizando los códigos generados.
5. **huffman\_decoding**: Decodifica el texto utilizando los códigos de Huffman.

**Ejemplo**:  
Para el texto "FACE", el programa genera códigos Huffman y los usa para codificar y luego decodificar el mensaje.

**APRENDIZAJES Y DIFICULTADES:**

**Aprendizajes**:

1. **Compresión de datos**: Aprendí cómo el algoritmo de Huffman puede reducir el tamaño de los datos, lo que es crucial para el procesamiento eficiente de grandes cantidades de información en áreas como redes y almacenamiento.
2. **Estructuras de datos**: La implementación del algoritmo me permitió entender mejor el uso de árboles binarios y montones (heaps) en la resolución de problemas complejos.
3. **Algoritmos de optimización**: Este algoritmo es un ejemplo claro de cómo los algoritmos pueden ser diseñados para optimizar recursos, algo que es esencial en el desarrollo de software eficiente.

**Dificultad**:  
La mayor dificultad fue entender cómo construir y recorrer el árbol de Huffman. Utilicé diagramas y pasos intermedios para visualizar cómo se combinan los nodos. También, al principio, tuve dificultades para implementar la decodificación correctamente, pero después de revisar el código y realizar pruebas, logré solucionarlo.

**ERRORES COMUNES DE DESARROLLADORES (EN INGLÉS):**

I watched the video about "Common Developer Mistakes" and here are the main errors discussed:

1. **Not understanding the problem**: Many developers jump into coding without fully understanding the requirements of the task. It's crucial to take time to analyze the problem before starting to code.
2. **Skipping documentation**: Developers often skip writing proper documentation for their code. This leads to confusion and errors later on, especially when other team members need to work with the code.
3. **Over-engineering solutions**: Some developers try to build overly complex solutions when a simpler approach would suffice. This increases development time and makes the code harder to maintain.
4. **Ignoring testing**: Not writing enough tests or ignoring testing practices can lead to bugs that are difficult to trace and fix.
5. **Failing to manage time effectively**: Poor time management can lead to rushing through tasks, which often results in subpar code or missed deadlines.

These are important lessons that I will apply in my future career, particularly in software development and project management, to avoid common pitfalls and deliver high-quality results.