Algoritmos y estructuras de datos.

Diseño de un algoritmo para identificar eficientemente a los 10 clientes más frecuentes dentro de un período de tiempo específico.

Estructura de conteo cliente frecuente:

El propósito es contar cuántas veces aparece cada cliente dentro del intervalo de tiempo usando una estructura dinámica (árbol binario de búsqueda).

```pseudocode
structure BSTNode:
    client_id
    count
    left
    right

function insertOrUpdate(BST, client_id):
    if BST is empty:
        return new BSTNode(client_id, 1)

    if client_id < BST.client_id:
        BST.left ← insertOrUpdate(BST.left, client_id)
    else if client_id > BST.client_id:
        BST.right ← insertOrUpdate(BST.right, client_id)
    else:
        BST.count ← BST.count + 1

    return BST
```

Consideraciones: Lo que propongo se realizará usando estructuras de datos clásicas.

James Montealegre Gutiérrez, Ingeniero Telemático, Universidad ICESI, Cali – Colombia
MSc. Ingeniería de Software, Universidad de los Andes, Bogotá – Colombia, Senior Fullstack Engineer.

Structure BSTNODE

Cada nodo representa un cliente único. El campo ***count*** almacena el número de veces que ha aparecido el cliente

Function insertOrUpdate(BST, client_id)

El árbol binario de búsqueda (BST) construye el orden de la estructura sobre la marcha. Usamos el client_id como clave de ordenación y almacenamos un contador como valor asociado. Cada que aparece un client_id, se ubica en el árbol (o se incrementa su contador si ya existe) no importa si el cliente aparece en posiciones aleatorias dentro del archivo, ya que el árbol garantiza la organización por client_id automáticamente.

¿Por qué es una buena opción para este problema?

A medida que recorremos transacciones, vamos actualizando los contadores sin tener que recorrer listas completas o usar estructuras costosas (Big O), además podemos procesar grandes volúmenes de datos mientras mantengamos la eficiencia del árbol (que este razonablemente balanceado.

En el contexto del problema planteado identificar los 10 clientes más frecuentes en un conjunto de transacciones dentro de un intervalo de tiempo determinado— se propone el uso de un Árbol Binario de Búsqueda (BST) como estructura principal para el conteo por cliente. Esta elección se justifica tanto desde el punto de vista de la eficiencia computacional como desde la claridad estructural propia de un enfoque basado en estructuras de datos clásicas.

Eficiencia en tiempo de ejecución

El BST permite realizar operaciones de inserción, búsqueda y actualización de manera eficiente, con una complejidad temporal promedio de O(log m), donde m es el número de clientes únicos en el intervalo evaluado. Esto representa una mejora considerable en comparación con estructuras secuenciales como listas simples, que tendrían una complejidad O(m) por operación.

A medida que se recorre cada transacción del archivo, si su marca de tiempo se encuentra dentro del rango solicitado, se actualiza o crea un nodo en el BST correspondiente al cliente.

James Montealegre Gutiérrez, Ingeniero Telemático, Universidad ICESI, Cali – Colombia
MSc. Ingeniería de Software, Universidad de los Andes, Bogotá – Colombia, Senior Fullstack Engineer.

Esta operación es directa y escalable, permitiendo el procesamiento eficiente de grandes volúmenes de datos sin degradación significativa del rendimiento.

Orden implícito y utilidad posterior

A diferencia de otras estructuras como los diccionarios o tablas hash, el BST mantiene los nodos organizados de acuerdo con la clave (en este caso, el client_id). Esta propiedad facilita el recorrido ordenado (inorden), útil para análisis adicionales o generación de reportes, y también permite construir estructuras derivadas como la lista de los 10 clientes más frecuentes.

Control de uso de memoria

El algoritmo no almacena la totalidad de las transacciones en memoria. En su lugar, únicamente mantiene un nodo por cliente en el BST, lo que permite trabajar con archivos que superen la memoria disponible. Esta estrategia es especialmente relevante considerando la condición del enunciado que requiere soportar grandes volúmenes de datos.

Enfoque estructurado y didáctico

La implementación de un árbol binario de búsqueda se ajusta a una enseñanza rigurosa de estructuras de datos, permitiendo abordar conceptos como recursión, recorridos en profundidad y eficiencia algorítmica. Su uso ofrece una solución equilibrada entre eficiencia y claridad, sin recurrir a herramientas externas o librerías avanzadas.

Comparación con alternativas

Otras estructuras como las tablas hash pueden ofrecer mayor velocidad promedio, pero sacrifican el orden implícito, y su gestión manual resulta menos transparente. Las listas simples, aunque fáciles de implementar, implican una penalización considerable en tiempo cuando se requiere búsqueda o actualización. Árboles balanceados como AVL o Red-Black Tree garantizan mejor comportamiento en el peor caso, pero a costa de una complejidad de implementación innecesaria para este escenario.

James Montealegre Gutiérrez, Ingeniero Telemático, Universidad ICESI, Cali – Colombia
MSc. Ingeniería de Software, Universidad de los Andes, Bogotá – Colombia, Senior Fullstack Engineer.

Top 10 de clientes:

Como parte del diseño del algoritmo, se requiere identificar los 10 clientes con mayor número de transacciones dentro de un rango de tiempo específico. Para cumplir este propósito, se propone utilizar una lista enlazada ordenada de tamaño fijo, en la que cada nodo contiene el identificador del cliente y su contador de transacciones. Esta estructura actúa como un contenedor del conjunto de resultados finales, y su diseño responde a criterios de simplicidad, eficiencia y control de recursos.

```pseudocode
structure ListNode:
    client_id
    count
    next

TopList ← empty

function insertOrdered(client_id, count):
    new_node ← create ListNode(client_id, count)
    insert new_node in descending order by count
    if size of TopList > 10:
        remove last node
```

James Montealegre Gutiérrez, Ingeniero Telemático, Universidad ICESI, Cali – Colombia
MSc. Ingeniería de Software, Universidad de los Andes, Bogotá – Colombia, Senior Fullstack Engineer.

```pseudocode
function insertOrdered(client_id, count):
    new_node ← create ListNode(client_id, count)

    if TopList is empty:
        TopList ← new_node
        return

    if count > TopList.count:
        new_node.next ← TopList
        TopList ← new_node
        return

    current ← TopList

    while current.next ≠ null AND current.next.count ≥ count:
        current ← current.next

    new_node.next ← current.next
    current.next ← new_node

    if size of TopList > 10:
        removeLastNode()
```

Estructura y funcionamiento

La lista se mantiene ordenada en forma descendente según el número de transacciones. Cada vez que se analiza un cliente durante el recorrido del árbol binario (estructura secundaria), se evalúa si su frecuencia es suficientemente alta para ingresar a la lista del top 10.

James Montealegre Gutiérrez, Ingeniero Telemático, Universidad ICESI, Cali – Colombia
MSc. Ingeniería de Software, Universidad de los Andes, Bogotá – Colombia, Senior Fullstack Engineer.

Si la lista tiene menos de 10 elementos, el nuevo cliente se inserta directamente en la posición correspondiente.

Si ya tiene 10 elementos, solo se inserta el nuevo cliente si su frecuencia es mayor que la del último nodo (el de menor frecuencia), en cuyo caso se elimina este último.

Esta lógica garantiza que en todo momento se mantenga una lista con los 10 clientes más frecuentes, sin necesidad de ordenar al finalizar.

Complejidad computacional

La complejidad de inserción en la lista está acotada, ya que su tamaño máximo es constante (10). En el peor de los casos, se recorre la lista completa para encontrar la posición de inserción, con una complejidad de O(10) = O(1). El control del tamaño también se realiza en tiempo constante.

Al procesar m clientes únicos extraídos del árbol binario, se realizan como máximo m operaciones de inserción o descarte en la lista, lo que implica una complejidad total de O(m).

Control de uso de memoria

La lista tiene un tamaño máximo conocido y constante. En todo momento, se almacenan como máximo 10 nodos, cada uno con un identificador y un contador. Esto hace que la estructura sea extremadamente liviana y predecible, sin importar la cantidad total de clientes procesados.

Además, al mantener únicamente los resultados necesarios, se evita el uso de estructuras adicionales para ordenamiento posterior (como arreglos temporales o estructuras de clasificación).

James Montealegre Gutiérrez, Ingeniero Telemático, Universidad ICESI, Cali – Colombia
MSc. Ingeniería de Software, Universidad de los Andes, Bogotá – Colombia, Senior Fullstack Engineer.

Validación de nodos en la lista top 10:

```pseudocode
function inOrderTraversal(node):
    if node is null:
        return

    inOrderTraversal(node.left)

    insertOrdered(node.client_id, node.count)

    inOrderTraversal(node.right)
```

Esta es una función recursiva que recorre un árbol binario de búsqueda (BST) inorden. Su propósito es visitar todos los nodos del árbol y, por cada nodo, evaluar si el cliente correspondiente debe ser insertado en la lista del top 10 de clientes más frecuentes.

Lógica principal:

```pseudocode
initialize BST ← empty
initialize TopList ← empty

open transaction file

while not end of file:
    read transaction (timestamp, client_id, amount)

    if timestamp is within [start_time, end_time]:
        BST ← insertOrUpdate(BST, client_id)

inOrderTraversal(BST)

return TopList
```

Análisis de complejidad temporal y espacial

Supuestos

N = Total de transacciones

K = Transacciones dentro del rango de fechas

M = Cantidad de clientes únicos en el rango M<=K

T= Tamaño máximo de la lista del top (10)

BigO (Time Complexity)

O(N) + O(K log M) + O(M) = O(N + K log M)

James Montealegre Gutiérrez, Ingeniero Telemático, Universidad ICESI, Cali – Colombia
MSc. Ingeniería de Software, Universidad de los Andes, Bogotá – Colombia, Senior Fullstack Engineer.

BigO (Space Complexity)

BST de clientes → O(M)

Lista del top → O(10) = O(1)

No se almacenan transacciones completas, se procesan en batch en flujo O(1)

Recursión inorden → O log(M)

O(M) + O(1) + O(log M) = O(M)

James Montealegre Gutiérrez, Ingeniero Telemático, Universidad ICESI, Cali – Colombia
MSc. Ingeniería de Software, Universidad de los Andes, Bogotá – Colombia, Senior Fullstack Engineer.