ESTRUCTURA DE DATOS II

Parcial #2 - Informe

En este documento se ver el análisis del código realizado para una Gestión de Productos de una Tienda en Línea

ALEC BIRUET
DANIEL ORTEGA
ESTEBAN DIMAS
RANDALL GUZMAN

ESTRUCTURA DE DATOS II

Parcial #2 - Informe

Resultados de ordenamiento

1. Insertion Sort

Insertion Sort ordena una lista construyéndola elemento por elemento. Toma un elemento de la lista no ordenada y lo inserta en el lugar correcto dentro de la parte ya ordenada. Es más eficiente que Bubble Sort en listas pequeñas o casi ordenadas, pero sigue teniendo una complejidad $O(n^2)$ en el peor de los casos.

```
 Ordenamientos por precio ascendente
Insertion Sort: 0.000054 s

 Ordenamientos por rating descendente
Insertion Sort: 0.000056 s
```

2. Bubble Sort

Bubble Sort es un algoritmo de ordenamiento simple que funciona comparando pares de elementos adyacentes y cambiándolos de posición si están en el orden incorrecto. Este proceso se repite varias veces hasta que la lista está completamente ordenada. Aunque es fácil de implementar, es poco eficiente para grandes cantidades de datos, ya que su complejidad es $O(n^2)$.

```
 Ordenamientos por precio ascendente
Bubble Sort:    0.000086 s

 Ordenamientos por rating descendente
Bubble Sort:    0.000091 s
```

3. Merge Sort

Merge Sort es un algoritmo eficiente y estable que sigue la estrategia de "divide y vencerás". Divide recursivamente la lista en mitades hasta que cada sublista tiene un solo elemento, y luego las combina (merge) de forma ordenada. Su complejidad es $O(n \log n)$ en todos los casos, lo que lo hace adecuado para trabajar con grandes volúmenes de datos en entornos de producción.

```
 Ordenamientos por precio ascendente
Merge Sort:     0.000050 s

 Ordenamientos por rating descendente
Merge Sort:     0.000045 s
```

Introducción

el objetivo del trabajo es comprender el uso correcto de los diferentes usos de algoritmos de ordenamiento y búsquedas

ESTRUCTURA DE DATOS II

Parcial #2 - Informe

ALGORITMO	RATING	PRECIO	CONCLUCIONES
Insertion Sort	0.000056 s	0.000054 s	Solo útil para listas pequeñas o propósitos educativos.
Bubble Sort	0.000091 s	0.000086 s	Rápido en listas pequeñas o casi ordenadas.
Merge Sort	0.000045 s	0.000050 s	Eficiente y consistente en listas grandes.

ESTRUCTURA DE DATOS II

Parcial #2 - Informe

Resultados de búsqueda

```
 BUSQUEDA BINARIA
Tiempo búsqueda binaria existentes:   0.000006 s
Tiempo búsqueda binaria inexistentes:   0.000006 s
```

¿Por qué la Búsqueda Binaria es ideal para buscar por id?

La búsqueda binaria es ideal para buscar por id porque los identificadores suelen ser valores únicos y pueden ordenarse fácilmente, lo que permite aprovechar la eficiencia de este algoritmo. La búsqueda binaria divide repetidamente el conjunto de datos ordenados en dos mitades, descartando la mitad en la que el elemento buscado no puede estar, lo que reduce drásticamente el número de comparaciones necesarias.

En un arreglo ordenado de n elementos, la búsqueda binaria tiene una complejidad de tiempo de O(log n), mucho más eficiente que la búsqueda lineal que es O(n). Esto la hace especialmente adecuada para bases de datos grandes, donde se necesita encontrar rápidamente un producto por su id único. Además, al ordenar previamente el arreglo por id, se garantiza que la búsqueda binaria pueda funcionar correctamente y de manera óptima.

ESTRUCTURA DE DATOS II

Parcial #2 - Informe

Reflexión sobre los Desafíos de Trabajar con Datos Complejos y la Elección de Algoritmos en Entornos de Producción

Análisis y Conclusiones

Trabajar con datos complejos, como objetos que representan productos con múltiples atributos, presenta varios desafíos que impactan directamente en el diseño y rendimiento de los algoritmos utilizados para su gestión. A diferencia de los datos simples, donde sólo se comparan valores primitivos, los datos complejos requieren considerar múltiples criterios de comparación, lo que añade complejidad a las operaciones de ordenamiento y búsqueda.

Uno de los principales desafíos es garantizar que los algoritmos manejen correctamente la comparación de objetos, especialmente cuando se deben ordenar por atributos distintos, como precio o calificación. Esto implica implementar funciones de comparación flexibles y eficientes, que puedan adaptarse a diferentes criterios sin afectar el rendimiento.

Además, el tamaño y la naturaleza dinámica de los datos influyen en la elección del algoritmo. Por ejemplo, con grandes volúmenes de datos, algoritmos con complejidades inferiores (como Quick Sort con O(n log n) en promedio) se vuelven necesarios para mantener tiempos de respuesta aceptables. En entornos de producción, la eficiencia no sólo afecta la velocidad, sino también el consumo de recursos, como memoria y procesamiento, impactando en la escalabilidad del sistema.

Otro aspecto importante es la facilidad de mantenimiento y la adaptabilidad del algoritmo. En sistemas reales, los requisitos pueden cambiar, y es vital que el código sea modular y permita incorporar nuevos criterios de ordenamiento o métodos de búsqueda sin grandes refactorizaciones.

Finalmente, en la toma de decisiones sobre qué algoritmo utilizar, es crucial balancear la complejidad teórica con la práctica real: entender el perfil de los datos, la frecuencia de operaciones, y las limitaciones del entorno técnico. Por ejemplo, para búsqueda por id, la búsqueda binaria es ideal cuando los datos están ordenados, mientras que para búsquedas más complejas por texto, puede requerirse implementar estructuras especializadas como índices o árboles de prefijos.