UNIVERSIDAD AMERICANA

Algoritmos y Estructuras de Datos

Metodos de Ordenamiento y Búsqueda

Algoritmo de Busqueda (Hashing) y Algoritmo de Ordenamiento (Bubble Sort)

Grupo 3

Integrantes:

- Jose Maria Moncada Maya
- Oscar Ivan Alvarado Jiron
- Oscar Enrique Arnuero Ramos

Docente:

- Silvia Gigdalia Ticay Lopez

Managua, 07 de Julio del 2025

1. Índice

1. **Introducción**
2. **Planteamiento del Problema**
3. **Objetivo de la Investigación**
4. **Objetivos Específicos**
5. **Metodología**
6. **Marco Conceptual**
   - **6.1. Hashing**
   - **6.2. Bubble Sort**
7. **Implementación del Algoritmo**
   - **7.1. Hashing (Bucket Sort / Tablas Hash)**
   - **7.2. Bubble Sort**
8. **Análisis a Priori**
   - **8.1. Análisis Hashing**
   - **8.2. Análisis Bubble Sort**
9. **Análisis a Posteriori**
   - **9.1. Análisis Hashing**
   - **9.2. Análisis Bubble Sort**
10. **Conclusión**
11. **Referencias Bibliográficas**

1. Introducción

En el ámbito de la ciencia de datos y la informática, la necesidad de optimizar la manipulación de información ha impulsado el desarrollo de estructuras eficientes para el almacenamiento y recuperación de datos. Uno de los métodos más destacados es el hashing, una técnica que permite transformar datos de entrada de longitud variable en salidas de longitud fija. El uso del hashing es común en áreas como estructuras de datos, criptografía, sistemas de autenticación y almacenamiento distribuido.

Sin embargo, su eficiencia depende de varios factores, como la calidad de la función hash y la gestión de colisiones. Ante esto, surge la necesidad de analizar detalladamente el comportamiento y eficiencia del algoritmo de hashing, tanto desde una perspectiva teórica (análisis a priori) como empírica (análisis a posteriori), considerando su desempeño en operaciones clave como búsqueda, inserción y acceso a datos.

En la actualidad, los algoritmos de búsqueda y ordenamiento son pilares fundamentales en el procesamiento de grandes volúmenes de datos. Mientras que las tablas hash (Hashing) permiten realizar búsquedas en tiempo casi constante, los algoritmos de ordenamiento, como Bubble Sort, son esenciales para preparar los datos antes de aplicar otras técnicas más avanzadas. En esta investigación, además de analizar la eficiencia de Hashing, se introduce el algoritmo Bubble Sort para comparar su comportamiento teórico y empírico. De esta forma, se resaltan las ventajas, limitaciones y contextos de uso de cada enfoque.

2. Planteamiento del problema

En el ámbito de la ciencia de datos y la informática, la necesidad de optimizar la manipulación de información ha impulsado el desarrollo de estructuras eficientes para el almacenamiento y recuperación de datos. Uno de los métodos más destacados es el hashing, una técnica que permite transformar datos de entrada de longitud variable en salidas de longitud fija. El uso del hashing es común en áreas como estructuras de datos, criptografía, sistemas de autenticación y almacenamiento distribuido.

Sin embargo, su eficiencia depende de varios factores, como la calidad de la función hash y la gestión de colisiones. Ante esto, surge la necesidad de analizar detalladamente el comportamiento y eficiencia del algoritmo de hashing, tanto desde una perspectiva teórica (análisis a priori) como empírica (análisis a posteriori), considerando su desempeño en operaciones clave como búsqueda, inserción y acceso a datos.

3. Objetivo de la Investigación

Analizar la eficiencia computacional del algoritmo de hashing mediante su implementación, evaluación y comparación con base en su comportamiento teórico y su rendimiento práctico en distintas condiciones de entrada.

4. Objetivos Específicos

- Comprender el funcionamiento interno del algoritmo de hashing y su aplicación en estructuras como tablas hash.
- Implementar funciones de hashing y técnicas de resolución de colisiones.
- Medir el tiempo de ejecución y uso de memoria del hashing en operaciones de inserción y búsqueda.
- Comparar el desempeño del hashing en su mejor y peor escenario.
- Evaluar la complejidad computacional usando notación Big O.

5. Metodología

5.1 Diseño de la investigación

El enfoque es experimental, ya que se diseñan escenarios controlados para observar el rendimiento del algoritmo de hashing al manipular estructuras de datos con diferentes volúmenes de información.

5.2 Enfoque de la investigación

Cuantitativo, pues se recogen datos objetivos y medibles (tiempo y memoria), utilizando herramientas como cronómetros de software y analizadores de memoria en Python.

5.3 Alcance de la investigación

Descriptivo y explicativo. Se describen las características del algoritmo de hashing y se explican sus comportamientos bajo diferentes condiciones de entrada.

5.4 Procedimiento

- Implementación de funciones hash y tablas hash con resolución de colisiones por encadenamiento.

- Ejecución de pruebas con conjuntos de datos de diferente tamaño.

- Medición del tiempo y memoria usados en operaciones de búsqueda e inserción.

- Análisis de los datos en relación con la complejidad teórica.

6. Marco Conceptual / Referencial

Un algoritmo es una secuencia finita de pasos lógicos que, al ser ejecutados, permiten resolver un problema específico (Cormen et al., 2009). En particular, el hashing es una técnica que aplica una función matemática a una clave de entrada para generar un índice, que indica la posición donde se almacenará o buscará el dato.

La notación Big O se utiliza para estimar la eficiencia algorítmica. Describe el crecimiento del tiempo de ejecución o del uso de memoria en función del tamaño de entrada n (Knuth, 1998). Los tipos más comunes de complejidad son:

- O(1): tiempo constante

- O(n): tiempo lineal

- O(n^2): tiempo cuadrático

Una tabla hash es una estructura de datos que utiliza hashing para mapear claves a posiciones. Su rendimiento se ve afectado por la calidad de la función hash y la estrategia de manejo de colisiones (como encadenamiento o direccionamiento abierto).

Bubble Sort es un algoritmo de ordenamiento sencillo basado en comparaciones sucesivas de pares adyacentes y el intercambio de elementos cuando están en el orden incorrecto. Aunque su eficiencia es baja para grandes volúmenes de datos, resulta útil para propósitos didácticos y casos con conjuntos pequeños.

La complejidad temporal de Bubble Sort es O(n^2) en el peor caso y en el caso promedio, mientras que su mejor caso (lista ya ordenada) es O(n). Su complejidad espacial es O(1), ya que ordena los elementos en el mismo arreglo.

n

7. Implementación del Algoritmo

Bubble Sort

```
def bubble_sort(arr):
    n = len(arr)
    # Recorremos todos los elementos del arreglo
    for i in range(n):
        # Últimos i elementos ya están en su lugar
        for j in range(0, n-i-1):
            # Intercambiamos si el elemento encontrado es mayor que el siguiente
```

```
            if arr[j] > arr[j+1]:

                arr[j], arr[j+1] = arr[j+1], arr[j]

    return arr



# Simulación de un problema real:

# Un negocio recibe pedidos con distintos tiempos de entrega estimados (en días).

# Se desea ordenarlos de menor a mayor para planificar las entregas.



print("----- PLANIFICADOR DE ENTREGAS -----")

entrada = input("Ingrese los tiempos estimados de entrega separados por coma (ej: 5,2,8,1,3): ")

# Convertimos la entrada en una lista de enteros

tiempos = [int(x.strip()) for x in entrada.split(',')]



print("\nTiempos de entrega recibidos:", tiempos)



# Ordenamos usando bubble sort

tiempos_ordenados = bubble_sort(tiempos.copy())



print("Tiempos de entrega ordenados:", tiempos_ordenados)



# Mostramos una planificación sencilla

print("\nPLAN DE ENTREGAS:")

for i, tiempo in enumerate(tiempos_ordenados, start=1):

    print(f"Día {i}: Entrega con tiempo estimado de {tiempo} días")
```

hash

```
# Implementación sencilla de tabla hash con encadenamiento


class TablaHash:

    def __init__(self, tamaño):

        self.tamaño = tamaño

        self.tabla = [[] for _ in range(tamaño)]


    def funcion_hash(self, clave):

        # Convierte la clave en un índice entre 0 y tamaño - 1

        return hash(clave) % self.tamaño


    def insertar(self, clave, valor):

        indice = self.funcion_hash(clave)

        # Revisar si la clave ya existe

        for i, (k, v) in enumerate(self.tabla[indice]):

            if k == clave:

                    self.tabla[indice][i] = (clave, valor)   # Actualiza
valor

                return

           self.tabla[indice].append((clave, valor))   # Si no existe,
agrega


    def obtener(self, clave):

        indice = self.funcion_hash(clave)

        for k, v in self.tabla[indice]:
```

```
            if k == clave:
                return v
        return None  # No encontrado

    def mostrar(self):
        for i, cubeta in enumerate(self.tabla):
            print(f"Índice {i}: {cubeta}")

# Uso del código
if __name__ == "__main__":
    tabla = TablaHash(5)

    # Insertar claves y valores
    tabla.insertar("nombre", "Eduardo")
    tabla.insertar("edad", 25)
    tabla.insertar("ciudad", "Managua")
    tabla.insertar("profesion", "Ingeniero")
    tabla.insertar("edad", 26)  # Actualiza valor existente

    # Mostrar tabla completa
    tabla.mostrar()

    # Obtener un valor
    print("Edad:", tabla.obtener("edad"))
```

8. Análisis a Priori

8.1 Eficiencia Espacial

La tabla hash ocupa espacio proporcional al número de casillas (buckets) y al número de colisiones. El uso de listas enlazadas para colisiones incrementa el consumo de memoria de forma controlada.

8.2 Eficiencia Temporal

- Hashing (sin colisiones): inserción y búsqueda en O(1).

- Hashing (peor caso con colisiones): inserción y búsqueda en O(n).

- Bubble Sort: peor caso y promedio O(n^2), mejor caso O(n).

8.3 Complejidad Algorítmica

La función hash simple (suma de caracteres) tiene complejidad O(m) donde m es la longitud de la clave. Bubble Sort requiere comparaciones sucesivas en un arreglo de tamaño n, resultando en O(n^2).

9. Análisis a Posteriori

9.1 Mejor Caso

Ocurre cuando no hay colisiones y cada clave accede a su propia casilla sin conflictos. El tiempo de búsqueda e inserción se mantiene en O(1)O(1).

9.2 Caso Promedio

Con claves aleatorias y una buena función hash, las colisiones son mínimas. El tiempo esperado sigue siendo cercano a O(1)O(1).

9.3 Peor Caso

Cuando todas las claves colisionan en la misma casilla, el tiempo de búsqueda se convierte en lineal, es decir, O(n)O(n), ya que hay que recorrer la lista enlazada completa.

9. Análisis a Posteriori

9.1 Mejor Caso

- Hashing: O(1) si no hay colisiones.

- Bubble Sort: O(n) si la lista ya está ordenada.

9.2 Caso Promedio

- Hashing: O(1) con buena función hash y distribuciones uniformes.

- Bubble Sort: O(n^2) en arreglos aleatorios.

9.3 Peor Caso

- Hashing: O(n) cuando todas las claves colisionan en la misma casilla.

- Bubble Sort: O(n^2) cuando el arreglo está en orden inverso.

10. Conclusión

En este estudio se analizaron dos técnicas fundamentales para el manejo de datos: Hashing y Bubble Sort. El hashing demuestra ser altamente eficiente para operaciones de búsqueda e inserción, siempre que la función hash y la gestión de colisiones sean adecuadas. Por otro lado, Bubble Sort, pese a su simplicidad y utilidad didáctica, no es recomendable para grandes volúmenes de datos debido a su alta complejidad temporal. Es importante seleccionar el algoritmo apropiado según las características del problema y los requisitos de rendimiento.

11. Referencias Bibliográficas

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley.

Stallings, W. (2017). *Cryptography and Network Security: Principles and Practice* (7th ed.). Pearson.

Weiss, M. A. (2011). *Data Structures and Algorithm Analysis in C++* (4th ed.). Pearson.

El Taller de TD. (2021). *Notación Big O | Explicación y Análisis de la complejidad de un Algoritmo* [Video]. YouTube. https://www.youtube.com/watch?v=rx7hQoPfiEo