UNIVERSIDAD ADOLFO IBÁÑEZ
FACULTAD DE INGENIERÍA Y CIENCIAS

Ayudantía 4 Estructura de datos

Profesores: Sebastián Sáez, Diego Ramos

Ayudantes: Diego Duhalde, Benjamín Wiedmaier, Fernando Zamora

1. Explique qué es una función hash y por qué es fundamental que ésta distribuya los datos de forma *uniforme* en una tabla hash. En su respuesta, mencione ejemplos de aplicaciones en el mundo real y los criterios que debe cumplir una "buena" función hash.

2. Considere la función hash $h(n) = n \mod 7$ y el conjunto de números $\{14, 3, 8, 22, 5, 16\}$.

   a. Inserte cada número en una tabla hash de 7 posiciones (inicialmente vacía) y escriba el estado de la tabla luego de cada inserción.

   b. ¿Qué sucede cuando se produce una colisión? Proponga una estrategia sencilla para resolverla.

3. Compare las listas enlazadas simples con los arreglos (vectores). En su respuesta, destaque las ventajas del dinamismo, la rapidez en inserción y borrado que presentan las listas, así como sus desventajas en cuanto a acceso aleatorio y uso de memoria.

4. Escriba en C el proceso para insertar un nodo al final de una lista enlazada simple.

5. Explique el funcionamiento del Selection Sort. ¿Cuál es la complejidad temporal en el peor caso y por qué?

6. Explique el algoritmo Merge Sort, haciendo énfasis en el proceso de división del arreglo y en la fusión ordenada de los subarreglos. Adicionalmente, dibuje (o describa) el árbol de recursión para un arreglo de 8 elementos.

7. Describa brevemente el funcionamiento de Quick Sort. ¿Por qué es importante la selección de un buen pivote y cómo puede afectar la complejidad temporal en el peor caso?

RESPUESTAS

1. Una función hash es un procedimiento que toma una entrada (o clave) y la transforma en un valor numérico, llamado hash, que se utiliza como índice en una tabla hash. Es fundamental que esta función distribuya los datos de forma uniforme para minimizar las colisiones, es decir, situaciones en las que dos claves diferentes producen el mismo hash.
   Una buena función hash debe ser rápida de calcular, generar valores distribuidos uniformemente y minimizar las colisiones. Ejemplos de aplicaciones incluyen sistemas de almacenamiento en caché, bases de datos y criptografía. Por ejemplo, en una base de datos, una función hash eficiente permite acceder rápidamente a los registros almacenados.

2. a. Estado de la tabla hash después de cada inserción:
      - Insertar 14: $14 \mod 7 = 0$, tabla: $[14, -, -, -, -, -, -]$
      - Insertar 3: $3 \mod 7 = 3$, tabla: $[14, -, -, 3, -, -, -]$
      - Insertar 8: $8 \mod 7 = 1$, tabla: $[14, 8, -, 3, -, -, -]$
      - Insertar 22: $22 \mod 7 = 1$, colisión en posición 1.
      - Insertar 5: $5 \mod 7 = 5$, tabla: $[14, 8, -, 3, -, 5, -]$
      - Insertar 16: $16 \mod 7 = 2$, tabla: $[14, 8, 16, 3, -, 5, -]$

   b. Cuando se produce una colisión, como en el caso de 22, se puede usar una estrategia de resolución como la **encadenamiento** (listas enlazadas en cada posición) o **sondeo lineal** (buscar la siguiente posición vacía en la tabla).

3. Las listas enlazadas simples y los arreglos (vectores) tienen diferencias significativas en su estructura y comportamiento. Las listas enlazadas son dinámicas, lo que significa que pueden crecer o reducirse en tamaño según sea necesario, mientras que los arreglos tienen un tamaño fijo una vez definidos.
   Además, las listas enlazadas permiten inserciones y eliminaciones rápidas en cualquier posición, ya que solo requieren ajustar los punteros, mientras que en los arreglos estas operaciones pueden ser costosas debido al desplazamiento de elementos.
   Sin embargo, los arreglos ofrecen acceso aleatorio a cualquier elemento en tiempo constante $O(1)$, mientras que en las listas enlazadas el acceso es secuencial, con una complejidad de $O(n)$. En términos de uso de memoria, las listas enlazadas requieren memoria adicional para almacenar los punteros, lo que puede ser una desventaja en comparación con los arreglos.

4. Revise la implementación del código en la rama de ayudantías en el repositorio del curso. Link.

5. El algoritmo de Selection Sort funciona seleccionando repetidamente el elemento más pequeño (o más grande, dependiendo del orden deseado) de la parte no ordenada del arreglo y colocándolo en su posición correcta en la parte ordenada. El proceso se repite hasta que todo el arreglo esté ordenado.

   El procedimiento es el siguiente:

   (a) Dividir el arreglo en dos partes: una parte ordenada (inicialmente vacía) y una parte no ordenada.

   (b) Encontrar el elemento más pequeño en la parte no ordenada.

(c) Intercambiar este elemento con el primer elemento de la parte no ordenada.

(d) Mover el límite entre las partes ordenada y no ordenada hacia la derecha.

(e) Repetir los pasos anteriores hasta que la parte no ordenada esté vacía.

La complejidad temporal en el peor caso es $O(n^2)$, donde n es el número de elementos en el arreglo. Esto se debe a que el algoritmo realiza $n-1$ comparaciones en la primera iteración, $n-2$ en la segunda, y así sucesivamente, lo que suma aproximadamente $\frac{n(n-1)}{2}$ comparaciones. Aunque es simple de implementar, no es eficiente para arreglos grandes.

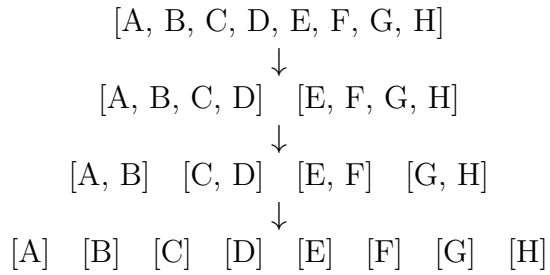
6. El algoritmo Merge Sort es un algoritmo de ordenamiento basado en el paradigma de divide y vencerás. Funciona dividiendo el arreglo en mitades hasta que cada subarreglo tenga un solo elemento (o ninguno), y luego fusionando estos subarreglos de manera ordenada para formar el arreglo final ordenado.

   El proceso se puede describir en los siguientes pasos:

   (a) Dividir el arreglo en dos mitades de manera recursiva hasta que cada subarreglo tenga un solo elemento.

   (b) Fusionar los subarreglos ordenados en un solo arreglo ordenado. Esto se realiza comparando los elementos de los subarreglos y seleccionando el menor (o mayor, dependiendo del orden deseado) en cada paso.

   El árbol de recursión para un arreglo de 8 elementos sería el siguiente:



   La complejidad temporal de Merge Sort es $O(n \log n)$ en el peor caso, ya que el arreglo se divide en $\log n$ niveles y en cada nivel se realizan $O(n)$ operaciones para fusionar los subarreglos.

7. El algoritmo Quick Sort es un método de ordenamiento basado en el paradigma de divide y vencerás. Funciona seleccionando un elemento como pivote y particionando el arreglo en dos subarreglos: uno con elementos menores al pivote y otro con elementos mayores. Luego, se aplica recursivamente el mismo procedimiento a los subarreglos.

   El proceso se puede describir en los siguientes pasos:

   (a) Elegir un pivote (puede ser el primer elemento, el último, uno aleatorio o el elemento medio).

   (b) Reorganizar el arreglo de manera que todos los elementos menores al pivote queden a su izquierda y los mayores a su derecha (este paso se llama partición).

   (c) Aplicar recursivamente Quick Sort a los subarreglos izquierdo y derecho.

La selección de un buen pivote es crucial para el rendimiento del algoritmo. Si el pivote divide el arreglo en partes aproximadamente iguales, la complejidad temporal promedio es $O(n \log n)$. Sin embargo, si el pivote es el elemento más pequeño o más grande en cada partición, el algoritmo degenera a $O(n^2)$ en el peor caso. Para evitar esto, se pueden usar estrategias como el pivote aleatorio o el pivote mediano de tres.