INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

Estructuras de Datos y Algoritmos

Act 5.1 - Implementación individual de operaciones sobre conjuntos

César Alán Silva Ramos

26/11/23

Complejidades computacionales

- quadratic

```
void HASH_TABLE::quadratic(int key) { // O(n)
  int index_hash = hash_tech(key); // O(1)
  if (table[index_hash].empty()) { // O(1)
     table[index_hash].push_back(key); // O(1)
  } else {
     int i = 1; // O(1)
     while (!table[(index_hash + i*i) % table.size()].empty()) { // O(n)
        i++; // O(1)
     }
     table[(index_hash + i*i) % table.size()].push_back(key); // O(1)
  }
}
```

En esta función, la mayoría de las operaciones tienen una complejidad constante, ya que se centran en verificar si una posición de la tabla está vacía o insertar una llave en una posición vacía, ambas con una complejidad de O(1). Sin embargo, la presencia de un bucle while introduce una complejidad distinta principal , este bucle busca una posición vacía, y en situaciones donde la tabla está completamente llena, se verá obligado a iterar a través de todos los elementos de la tabla. Esta iteración se vuelve directamente proporcional al tamaño de la tabla, generando así una complejidad **temporal de O(n)**.

En términos de complejidad espacial, la función mantiene un rendimiento constante, ya que no requiere estructuras de datos adicionales que crezcan con el tamaño de la entrada. Por lo tanto, la complejidad **espacial** sigue siendo **O(1)**, indicando que el uso de memoria adicional no aumenta proporcionalmente al tamaño de la entrada.

- chain

```
void HASH_TABLE::chain(int key) { // O(1)
  int index_hash = hash_tech(key); // O(1)
  table[index_hash].push_back(key); // O(1)
}
```

Esta función cuenta con una complejidad temporal constante, expresada como O(1). Esta característica se debe a dos operaciones clave: una que calcula y devuelve un valor, cuyo tiempo de ejecución siempre es constante, y otra que realiza la inserción en la tabla, también constante. La combinación de estas dos operaciones resulta en una complejidad **temporal** final de O(1) + O(1) = **O(1)**. En términos prácticos, esto significa que el rendimiento de la función no varía con el tamaño de la entrada, manteniendo su eficiencia constante.

En relación con la complejidad espacial, se alcanza un rendimiento constante, expresado como O(1). Esto se debe a que, en la implementación común de push_back, que agrega elementos al final de la lista, el espacio en memoria se considera constante. Además, gracias a la implementación específica del vector que se utilizó, la complejidad **espacial** se mantiene en **O(1)**.

- disp_tab

```
void HASH_TABLE::disp_tab() {
  for (const auto& lista : table) {
        if(lista.empty()){
           std::cout<<"null"<<std::endl;
        }
        else{
           for (const auto& elemento : lista) {
           std::cout << elemento << " ";
           }
           std::cout << std::endl;
        }
    }
```

```
    std::cout << std::endl;
}
```

La complejidad temporal de esta función se ve influenciada por el número total de elementos en la tabla hash y el tamaño de las listas que sirven como contenedores. Supongamos que hay N elementos en total, y M representa el tamaño promedio de los contenedores. En este escenario, la iteración a través de todos los elementos ocurre una vez, lo que implica una complejidad de O(N) en el peor de los casos. Además, para cada contenedor, la iteración sobre sus elementos tiene una complejidad de O(M) Dado que estas operaciones son secuenciales, la complejidad total se expresa como O(N * M) .No obstante, si M se mantiene constante, es decir, si el tamaño promedio de los contenedores no depende de N, entonces la complejidad temporal se reduce a O(N). La complejidad espacial de O(1) viene de la presencia de variables que no aumentan proporcionalmente al tamaño de la entrada. Esta característica demuestra la independencia de la memoria utilizada con respecto al tamaño de la entrada.

Casos prueba

Se emplearon dos tablas de hash para implementar dos enfoques distintos de manejo de colisiones: uno basado en la resolución cuadrática y otro en el encadenamiento.

Caso 1

Input

```
int hash_nums[] = {5,16,14,78,52,50,36,58,10};
```

Output con estrategia cuadrática

```
-- Quadratic Strategy applied to hash table ---
50
10
52
null
14
5
16
36
78
58
```

Caso 2

Input

```
int hash_nums[] = {5,16,14,78,52,50,36,58,10};
```

Output con estrategia de encadenamiento

```
--Chain Strategy applied to hash table ---
50 10
null
52
null
14
5
16 36
null
78 58
null
```

Caso 3

Input

```
int hash_nums[] = {14,32,16,18,52,20,30};
```

Output con estrategia cuadrática

```
-- Quadratic Strategy applied to hash table ---
20
30
32
52
14
null
16
null
18
null
```

Caso 4

Input

```
int hash_nums[] = {14,32,16,18,52,20,30};
```

Output con estrategia de encadenamiento

```
--Chain Strategy applied to hash table ---
20 30
null
32 52
null
14
null
16
null
18
null
```