ACT_5.1_OPERACIONES_SOBRE_CONJUNTOS

Casos de prueba:

Antes de mostrar los casos de prueba cabe resaltar que se utilizaron dos tablas con una longitud de 10 valores para llevar a cabo la técnica escogida de hashing y por su parte, para implementar las estrategias de control de colisiones en estas mismas.

Se implementaron dos principales estrategias de control de colisiones: cuadrática y de encadenamiento.

1) Que los datos ingresados con la técnica de hashing en la tabla de 10 espacios, se acomoden acorde a la estrategia de control de colisiones utilizada.

Estrategia: Cuadrática.

Input:

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

Output:

```
--MODIFIED HASH TABLE 1 WITH QUADRATIC STRATEGY ---
50
10
52
NULL
14
5
16
36
78
58
```

2) Que los datos ingresados con la técnica de hashing en la tabla de 10 espacios, se acomoden acorde a la estrategia de control de colisiones utilizada.

Estrategia: Encadenamiento.

Input:

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

Output:

```
--MODIFIED HASH TABLE 2 WITH CHAIN STRATEGY ---
50 10
NULL
52
NULL
14
5
16 36
NULL
78 58
NULL
```

3) Que los datos ingresados con la técnica de hashing en la tabla de 10 espacios, se acomoden acorde a la estrategia de control de colisiones utilizada.

Estrategia: Cuadrática.

Input:

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

Output:

```
--MODIFIED HASH TABLE 1 WITH QUADRATIC STRATEGY ---
20
30
32
52
14
NULL
16
NULL
18
NULL
```

4) Que los datos ingresados con la técnica de hashing en la tabla de 10 espacios, se acomoden acorde a la estrategia de control de colisiones utilizada.

Estrategia: Encadenamiento.

Input:

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

Output:

```
--MODIFIED HASH TABLE 2 WITH CHAIN STRATEGY ---
20 30
NULL
32 52
NULL
14
NULL
16
NULL
18
NULL
```

Complejidades Computacionales:

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 computacional de esta función es de O (n^2), ya que, contamos con dos bucles for anidados y por su parte siempre que queramos imprimir nuestra tabla debemos de recorrer todo el vector y a su vez todas las listas que conforman a este mismo.

2) quadratic ():

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

La complejidad computacional de esta función es de O (n), ya que, en el peor de los casos el bucle while tiene que recorrer todos los elementos de la tabla. Es decir, que en el peor de los casos se tendrán que recorrer los n elementos de la tabla.

3) chain ():

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

La complejidad computacional de esta función es de O (1), debido a que, simplemente contamos con líneas de código sencillas que solo llevan a cabo funciones independientes. No contamos con ningún bucle ni ninguna otra cosa que pueda afectar la complejidad de nuestra función.

4) hash_tech():

```
int HASH_TABLE::hash_tech(int key) {
    return key % table.size();
}
```

La complejidad computacional de esta función es de O (1), debido a que, simplemente contamos con líneas de código sencillas que solo llevan a cabo funciones independientes. No contamos con ningún bucle ni ninguna otra cosa que pueda afectar la complejidad de nuestra función.