**Análisis de complejidad temporal**

i) add(E element) [Class CustomLinkedList]

1 public void add(E element) {

2 Node<E> newNode = new Node<>(element);

3 newNode.next = head;

4 head = newNode;

5 size++;

6 }

La línea Node<E> newNode = new Node<>(element) tiene una complejidad temporal O(1) (constante), porque sólo asigna e inicializa un objeto.

La línea newNode.next = head tiene una complejidad temporal O(1) (constante), porque sólo asigna un nuevo valor a un objeto.

La línea head = newNode tiene una complejidad temporal O(1) (constante), porque sólo asigna un nuevo valor a un objeto.

La línea size++ tiene una complejidad temporal O(1) (constante), porque sólo asigna un nuevo valor a un objeto.

Por ende, la complejidad temporal del algoritmo es O(1) (constante).

ii) get(int index) [Class CustomLinkedList]

1 public E get(int index) {

2 if (index < 0 || index >= size) {

3 throw new IndexOutOfBoundsException("Index out of bounds");

4 }

5 Node<E> current = head;

6 for (int i = 0; i < index; i++) {

7 current = current.next;

8 }

9 return current.data;

10 }

Las líneas 2 - 4 sólo hacen una comparación, por lo que su complejidad es O(1).

La línea 5 sólo asigna un valor a un objeto, por lo que su complejidad es O(1).

Las líneas 6 y 7 ocurren dentro de un for, por lo que su complejidad es, en el peor de los casos, O(N).

La línea 10 es un return, por lo que su complejidad es O(1).

Por ende, la complejidad temporal del algoritmo es O(N) (lineal).

**Análisis de complejidad espacial**

i) add(E element) [Class CustomLinkedList]

(mismo código)

El método sólo asigna un nuevo espacio de memoria para el nuevo nodo, por lo que su complejidad es O(1) (el aumento es constante porque sólo asigna memoria para un nuevo elemento).

Por ende, la complejidad temporal del algoritmo es O(1) (constante).

ii) resize() [Class CustomHashMap]

1 private void resize() {

2 Entry<K, V>[] oldTable = table;

3 table = new Entry[table.length \* 2];

4 size = 0;

5

6 for (Entry<K, V> entry : oldTable) {

7 while (entry != null) {

8 put(entry.key, entry.value);

9 entry = entry.next;

0 }

1 }

2 }

La línea 2 asigna un espacio de memoria a una nueva variable, oldTable, por lo que su complejidad es O(N) (proporcional a CustomHashMap.size)

La línea 3 asigna un espacio de memoria a la variable table, también proporcional a CustomHashMap.size, por lo que su complejidad es, igualmente, O(N).

La línea 4 del método sólo modifica el valores de una variable ya designada previamente, por lo que su complejidad es O(1).

Tanto el for como el while iteran por cada elemento en los Entries de table, pero no designan más memoria (toda la memoria que iba a alocarse fue designada en la línea 3), por lo que su complejidad espacial es O(1).

Por ende, la complejidad espacial del método es O(N) (lineal).

**Ambos,** porque me di cuenta de que la rúbrica pide un método largo

iii) remove(K key) [Class CustomHashMap]

1 public V remove(K key) {

2 int hash = hash(key);

3 int index = getIndex(hash);

4

5 Entry<K, V> prev = null;

6 Entry<K, V> current = table[index];

7

8 while (current != null) {

9 if (current.key.equals(key)) {

0 if (prev == null) {

1 table[index] = current.next;

2 } else {

3 prev.next = current.next;

4 }

5 size--;

6 return current.value;

7 }

8 prev = current;

9 current = current.next;

0 }

1

2 return null;

3 }

El método es largo, pero la complejidad es baja, porque la única línea cuya complejidad NO es constante es la línea 8 (while), cuya complejidad temporal es, en el peor de los casos, O(N), lo que la haría lineal. No obstante, incluso en esos casos, el HashMap debería ser capaz de evitar (o al menos reducir) las colisiones de tal manera que el while no llega a recorrer muchos nodos, haciendo su complejidad temporal (para fines prácticos) casi constante (sigue siendo lineal, al menos para propósitos de calificar este ejercicio). Y la cantidad de espacio que ocupa es constante, por lo que la espacial es O(1).