```
public void insert(K key, V value) throws DuplicateItemException{
        int i = 0;
        do {
            int j = h(key, i);
            if (table[j] == null) { // si la posición está libre
                table[j] = new Node<>(key, value);
                return;
            } else {
                if (!(table[j].getKey() == key)){
                    i++;
                }else {
                    //Elemento duplicado
                    throw new DuplicateItemException();
                }
            }
        } while (i != arrSize); // si se ha recorrido toda la tabla, hay desbordamiento
        throw new RuntimeException("Desbordamiento de tabla hash");
    }
```

Time Complexity:

The spatial complexity of this algorithm is O(N), where N is the size of the hash table. This is because the hash table is stored in an array of size N and up to N elements can be stored in the table.

Explanation:
For the assignment of the variable i to 0, the operation is just one operation and it is performed in constant time, therefore it is O(1); When it enters the do-While loop, it is the same as the previous one, it only has a single operation and it is carried out in constant time, that is, the same as O(1). For the calculation of the index j of the hash table, implemented the hash function h(key, i), for this operation it depends on the implementation of the hash function, but it is also a constant complexity therefore it is also O(1) in the worst case.
For checking if the position of hash table[j] is free: This operation also has constant O(1) complexity. If the position is free, a new node is created and assigned to the hash table at position j: This operation also has a constant complexity O(1). If the position is not free, it checks if the element already exists in the hash table, the complexity of this operation is constant O(1) because the key is directly compared of the element in the hash table with the key to insert.
If the element exists in the hash table, an exception is thrown: This operation also has constant O(1) complexity.
If the element does not exist in the hash table, increment i by 1. This operation also has constant O(1) complexity. It checks if the entire hash table has been traversed, this operation also has constant O(1) complexity. If the entire hash table has been traversed and no free position has been found, an exception is thrown: This operation also has a constant O(1) complexity.
The execution time of the algorithm depends on the number of iterations of the do-while loop, which is limited by the size of the hash table. In the worst case, when the hash table is

completely full and there are no free positions, the algorithm will have to traverse the entire hash table before throwing an exception. Therefore, the time complexity of the algorithm is O(N), where N is the size of the hash table.

```
public void insert(K key, V value) throws DuplicateItemException{
        int i = 0;
        do {
            int j = h(key, i);
            if (table[j] == null) { // si la posición está libre
                table[j] = new Node<>(key, value);
                return;
            } else {
                if (!(table[j].getKey() == key)){
                    i++;
                }else {
                    //Elemento duplicado
                    throw new DuplicateItemException();
                }
            }
        } while (i != arrSize); // si se ha recorrido toda la tabla, hay desbordamiento
        throw new RuntimeException("Desbordamiento de tabla hash");
    }
```

Complejidad Espacial:

Tipo	**Variable**	**Tamaño**	**Cantidad de valores Átomicos**
Entrada	**K** **V**		**n** **1**
Auxiliar	**i** **j**		**1** **1**

- **Complejidad Espacial Total= Entrada + Auxiliar = n + 3 =** $\ominus\ (n)$
- **Complejidad Espacial Auxiliar= 1 + 1 =** $\ominus\ (1)$