**Temporal Complexity Analysis of the put Method in the Hash Table Class**

The **put** method performs the following operations:

**1.** **Key Verification (key == null):**This operation is a simple comparison that takes constant time O(1).

**2. Hash Calculation (int hash = hash(key)):**Assuming that the `hash(key)` function has constant complexity O(1), this step also takes constant time.

**3.** **Access and Modification of the Hash Table (table[hash]):** Accessing a specific position in the hash table takes constant time O(1).

**4. Key Search in the Corresponding List (table[hash].searchNode(...)):** The complexity of searching in a linked list can be linear with respect to the length of the list in the worst case. So, if the list has n elements, the search could take O(n) time in the worst case.

**5. Adding a New Element to the List (table[hash].addNode(...)):** Inserting an element into a linked list generally takes constant time O(1), as it simply involves adjusting pointers.

Therefore, the total complexity of the `put` method would be the sum of the complexities of each step:

- Key verification and hash calculation: O(1)

- Access and modification of the hash table: O(1)

- Key search in the corresponding list: O(n) in the worst case

- Adding a new element to the list: O(1)

The total complexity of the put method would indeed be O(1) + O(1) + O(n) + O(1), which simplifies to O(n) in the worst case.The temporal complexity of the `put` method is linear with respect to the number of elements in the corresponding list of the hash table, which is represented as O(n) in the worst case.

**Temporal Complexity Analysis of the `shuffle` Method in the ShufflableStack Class**

The `**shuffle**` method performs the following operations:

**1. Creation of the temporary list (List<T> tempBuffer = new ArrayList<>()):** Creating an empty temporary list has a constant time complexity of O(1).

**2. While loop (while (!isEmpty()) { ... }):** This loop iterates until the stack is empty. In each iteration, a `pop()` operation is performed, which has a time complexity of O(1) on a stack. Therefore, if the stack contains n elements, this loop would have a complexity of O(n) in the worst case.

**3. Adding elements to the temporary list (tempBuffer.add(pop())):** Adding an element to a dynamic list has an amortized complexity of O(1) on average, but since it's being done within a loop that runs n times (where n is the size of the stack), the total complexity would be O(n).

**4. Shuffling the list (Collections.shuffle(tempBuffer, rand))**: The shuffle function of the Collections class uses a random shuffle algorithm with a linear time complexity of O(n), where n is the size of the list.

**5. For loop (for (T item : tempBuffer) { ... }):** This loop iterates over all elements of the temporary list, which contains n elements. Therefore, it has a complexity of O(n).

**6. push() operation inside the for loop:** The push() operation has a time complexity of O(1) on a stack. Since the loop runs n times, the total complexity is O(n).

**1. Creation of the temporary list (List<T> tempBuffer = new ArrayList<>()):**O(1)

**2. While loop (`while (!isEmpty()) { ... }):** O(n)

**3. Adding elements to the temporary list (tempBuffer.add(pop())):** O(n)

**4. Shuffling the list (Collections.shuffle(tempBuffer, rand)):** O(n)

**5. For loop (for (T item : tempBuffer) { ... }):** O(n)

**6. push() operation inside the for loop:** O(n)

Adding up all these complexities, we get:

O(1) + O(n) + O(n) + O(n) + O(n) + O(n) = O(1 + 5n) = O(n)

The total temporal complexity of the `shuffle` method would be dominated by the list shuffling operation, which is O(n). Therefore, the overall temporal complexity of the `shuffle` method is O(n), where n is the size of the stack.