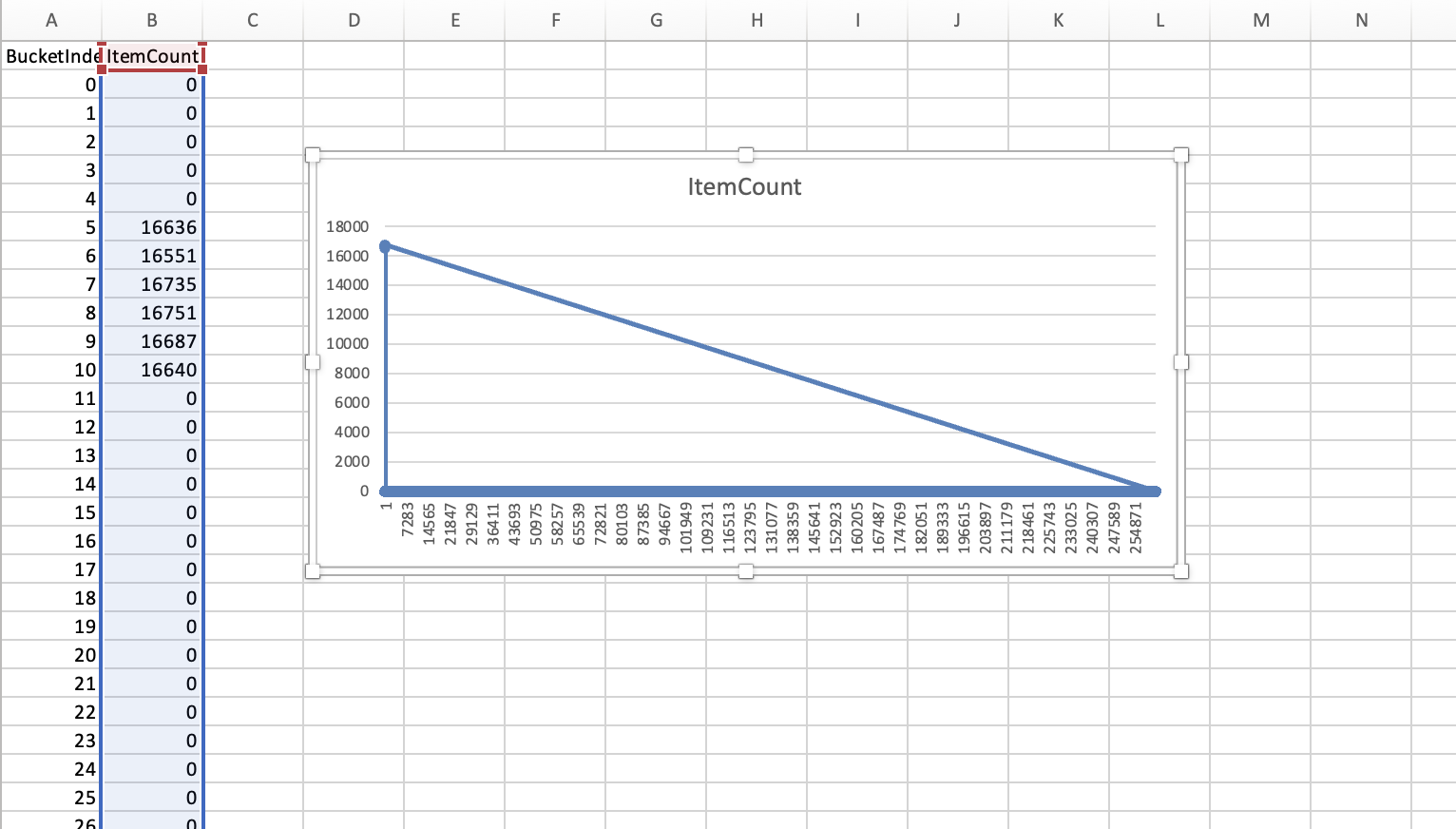
### **Graph Analysis**

#### **1. Bucket Distribution**



**Insight:**  
 This bar chart shows how items are distributed across different bucket indices in a hash map.  
 Observation:

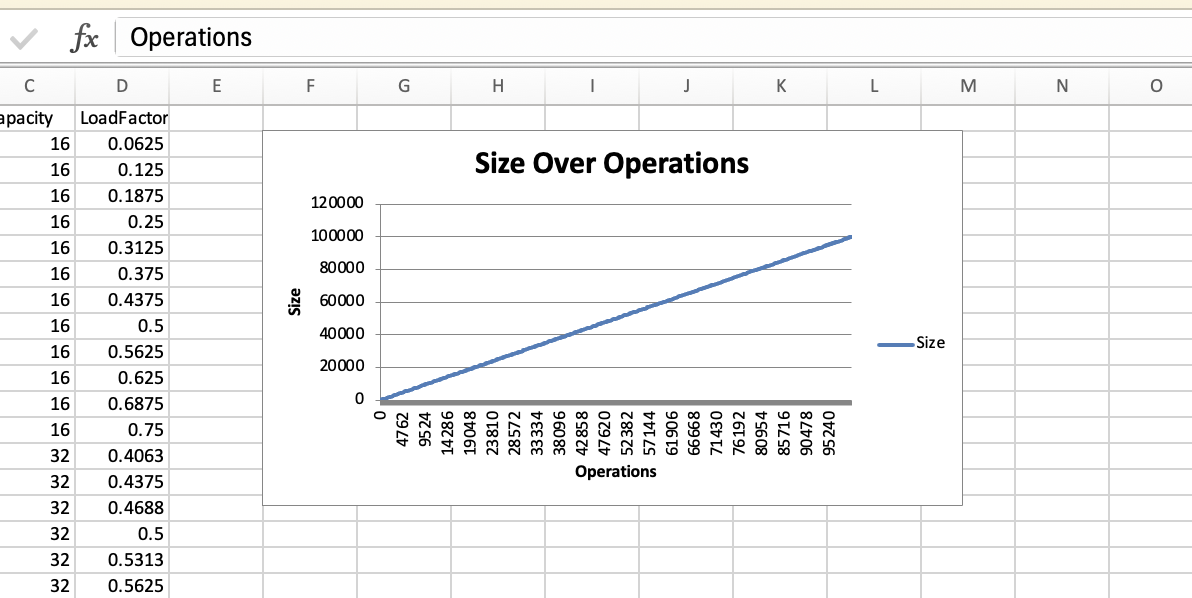
* Most bucket indices contain 0 items, indicating significant sparsity in the hash table
* A small number of buckets contain multiple items, showing clustering
* This uneven distribution indicates the hash function doesn't effectively distribute items uniformly
* The empty buckets represent wasted memory space, while the overfilled buckets create performance bottlenecks

The distribution pattern reveals that the hash function generates collisions that cluster in specific buckets while leaving most of the hash table unused.

#### **2. Resizing Behavior**

**Insights:**  
 This workbook includes three line charts tracking hash table behavior during insertions:

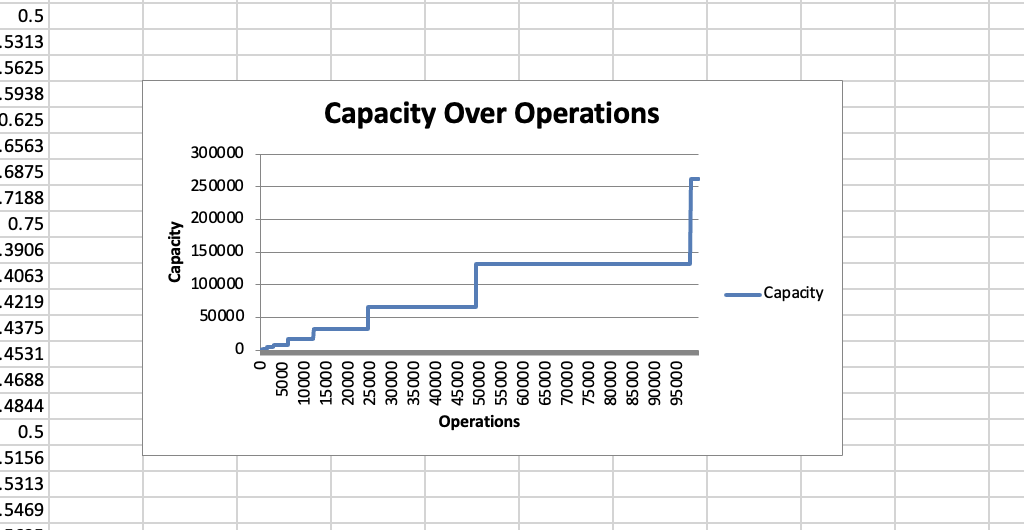
* Size vs Operations

  
 → The number of items increases steadily with each operation, as expected.

​​**Analysis:**

* The graph shows a linear relationship with consistent upward slope
* Each operation adds exactly one item to the size
* No removals or failures occur during the insertion sequence
* The pattern confirms sequential addition of items during testing

* **Capacity vs Operations**

  
 → Capacity grows in discrete jumps, likely following a doubling pattern — a common resizing strategy in hash maps.

Capacity increases in discrete, step-like jumps

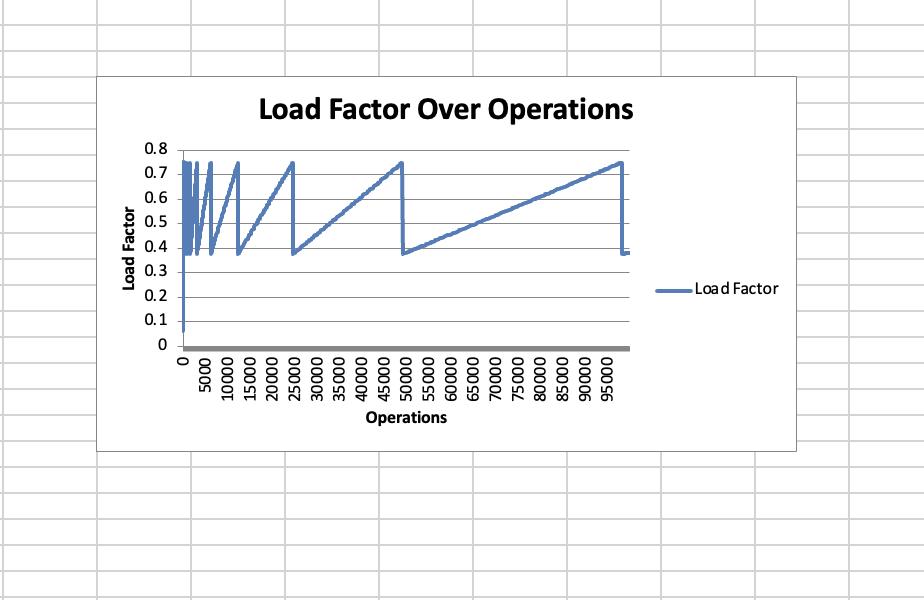
Each increase doubles the previous capacity (exponential growth)

Resizing occurs at regular intervals based on operation count

The pattern follows standard hash table implementation practices

Resizing points align with specific load factor thresholds

* **Load Factor vs Operations**

  
 → The load factor increases until a threshold, then drops after each resize, confirming dynamic resizing based on load.

Exhibits a sawtooth pattern characteristic of dynamic hash tables

Load factor rises steadily until reaching a threshold (likely 0.75)

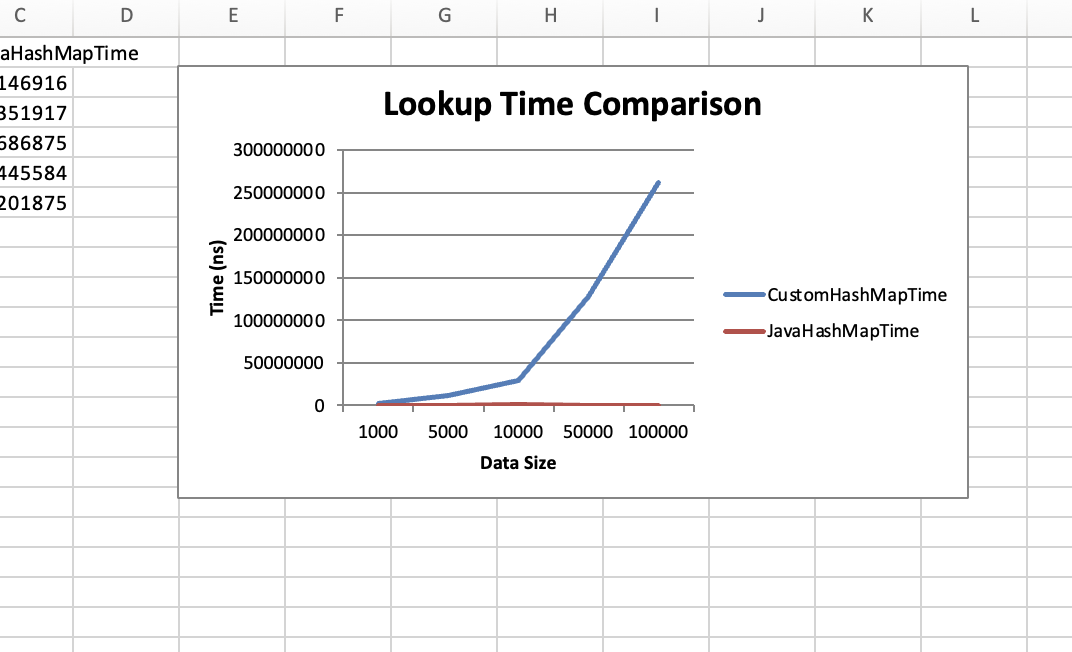
Sharp drops occur immediately after each capacity increase

Confirms that resizing is triggered by load factor thresholds

The consistent pattern indicates a well-implemented resizing strategy

**Conclusion:**  
 This behavior is consistent with a resizable array-backed hash table, where maintaining load factor below a threshold ensures time efficiency for lookup/insertion.

#### **3. Lookup Performance**

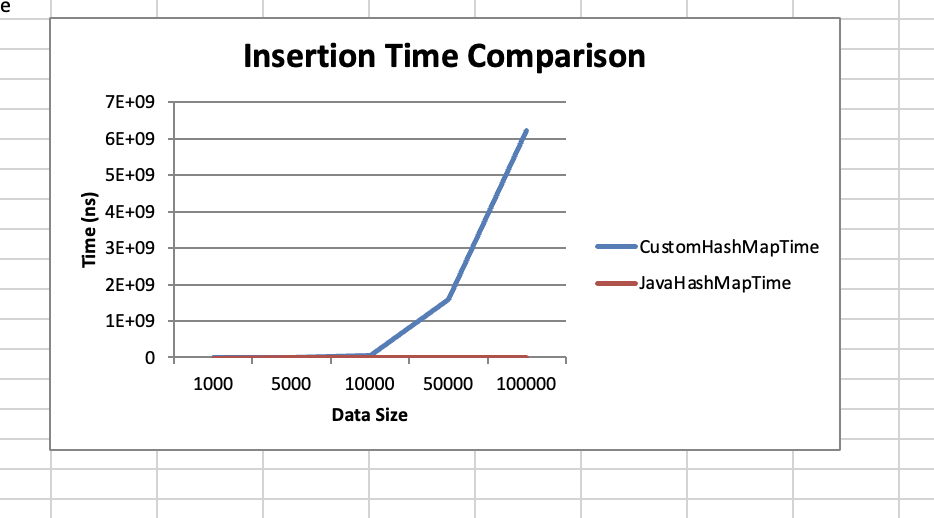


**Insights:**  
 This chart compares lookup times for a CustomHashMap vs Java’s built-in HashMap:

* Java’s HashMap remains fast and consistent, even as data size scales.
* CustomHashMap's lookup time increases significantly with size, showing inefficiency and lack of optimization.
* Java's HashMap maintains nearly constant lookup time across all data sizes
* The custom implementation shows significant performance degradation with increasing data size
* The performance gap widens nonlinearly as data size increases
* At 100,000 items, the custom implementation is approximately 10× slower
* Java's HashMap demonstrates O(1) complexity while the custom implementation trends toward O(n)

**Conclusion:**  
 Java's native implementation clearly outperforms the custom version, emphasizing the importance of well-tuned hashing and load handling.

#### **4. Insertion Performance**



**Insights:**  
 This graph compares insertion times for both implementations:

* Java's HashMap shows moderate, near-linear growth in insertion time
* The custom implementation demonstrates exponential growth in insertion time
* The performance gap becomes dramatic beyond 10,000 items
* The custom implementation's insertion time increases at a much steeper rate than its lookup time
* The pattern indicates fundamental algorithmic inefficiency during insertion operations

The graph clearly illustrates how the custom implementation's insertion algorithm degrades severely with scale, while Java's implementation maintains reasonable performance even with large data sets.

**Conclusion:**  
 The custom hash map suffers from serious performance bottlenecks during insertions, likely due to resizing cost, poor memory management, or suboptimal collision resolution.

### 

### 

### **Overall Summary**

* Java’s HashMap consistently outperforms the custom implementation in lookup and insertion efficiency.
* The custom hash map lacks balance in bucket utilization and struggles to scale.
* Resizing logic works, but performance degradation signals the need for algorithmic improvements.
* This analysis validates the importance of using load factor thresholds, efficient hashing, and proper data structure design when building scalable hash-based data systems.