Collision Resolution Hash Tables: A Comparative Performance Study Using Synthetic Data in C++

Dambo Itari, Obhuo Benjamin and Oguara Richard
Department of General Studies, College of Health Technology, Bayelsa State, Nigeria
Email: anidam43@gmail.com, Benjao2003@yahoo.com, oguararichard02@gmail.com
Phone: +2348035106767, +2347016467641, +2347036972948
DOI: 10.56201/ijcsmt.vol.11.no5.2025.pg34.45

Abstract
Hash tables are essential for fast data storage and retrieval; however, managing collisions remains a core challenge that affects overall efficiency. This research aims to evaluate and compare the performance of three prominent collisions resolution strategies; Linear Probing, Quadratic Probing, and Double Hashing in hash tables. To achieve this, C++ implementation was developed, and synthetic datasets of varying sizes were generated to simulate different load conditions. The research employed experimental method, measuring key performance indicators such as execution time, number of collisions and memory consumption across multiple trials were employed to assess performance under varying load factors.
The results reveal that Linear Probing is prone to primary clustering leading to significant performance degradation as the load factor increases. Quadratic Probing mitigates clustering more effectively but encounter limitations when its probing sequence cycles. Double Hashing consistently outperforms the other techniques, delivering superior results through more uniform distribution of keys, particularly in high-load environments.
This study concludes that Double Hashing offers the best balance of speed and efficiency for collision resolution, making it a preferred choice for optimizing hash table performance in data-intensive and high-performance computing applications.

Keywords: *Hash Tables, Collision Resolution, Linear Probing, Quadratic Probing, Double Hashing, C++ Implementation*

1. INTRODUCTION

Hashing is a computational technique that transforms input data into a fixed-size numerical value, known as a hash code, using a mathematical function called a hash function. This process allows data to be efficiently indexed and retrieved, particularly in a structure known as a hash table.

A hash table, also called a hash map, is a key-value data structure used for fast data access. It stores data in an array format, where each position, referred to as a bucket, is accessed via a hash code derived from the key. Hash tables are widely utilized in applications such as database indexing, caching systems, and network routing because of their average-case time complexity of O(1) for insertion, deletion, and search operations (Comen et al., 2009).

However, one of the major issues encountered in hash tables is collision—a scenario where two or more keys are assigned to the same bucket. Handling such collisions effectively is essential to maintaining the performance of the data structure.

Hashing is also often used to implement data structures like dictionaries. It offers one of the fastest mechanisms for data retrieval, bypassing the need for sequential or index-based searches by calculating the direct memory location of the data.

A hash table maps keys to values using an underlying array of fixed size, with index positions ranging from 0 to m - 1. Each position or bucket in this array stores a value, and the location is determined by a hash function.

A hash function is a mathematical formula used to convert keys into hash codes. These codes act as indices for storing values in a hash table. For example, using the division method, a hash function is expressed as:

f(k) = k mod m

where: k is the key

m is the size of the hash table

% is the modulo operator

If key 90 is hashed using f(90) = 90 mod 9, the resulting index is 0. Similarly, f(58) = 58 mod 7 = 2, placing the value at index 2.

Collision occurs when multiple keys are mapped to the same index. For example, hashing 66 using f(66) = 66 mod 8 = 2 may conflict with an existing value already stored at index 2.

To resolve collisions, several open addressing methods are employed:

- Linear Probing: Sequentially checks the next available index. Quadratic Probing: Uses a non-linear interval to reduce clustering.
- Double Hashing: Utilizes a second hash function to compute probe sequences and avoid primary clustering.
- Double Hashing: Utilizes a second hash function to compute probe sequences and avoid primary clustering.

Illustration of Linear Probing:

Using h(x) = x % 7, if inserting 30, it maps to slot 2. If the next key, say 29, also hashes to an occupied index, the algorithm checks the next slots until a free one is found.

Quadratic Probing uses the formula:

$h(k, i) = (h(k) + i^2) \bmod n$

Double Hashing employs:

Index = (h1(k) + i * h2(k)) mod table size

Synthetic data are artificially generated datasets designed to replicate the statistical properties, structural patterns, and relationships of real-world data, yet without containing any actual personal, sensitive, or identifiable records. Within computing and machine learning, such data are typically created through techniques like generative adversarial networks (Lu et al., 2023). In C++, synthetic data can be generated using custom algorithms that produce values based on specific statistical distributions, allowing precise control over load factors and collision rates.

2. RELATED WORKS

Hashing is a fundamental technique in computer science used to map data to a fixed-size value (hash code) for efficient storage and retrieval. Collision resolution is a critical aspect of hashing; as different keys may produce the same value. Various strategies such as linear probing, quadratic probing, and double hashing are employed to handle collisions.

Collision resolution is a fundamental challenge in hash table implementation, significantly affecting the efficiency of data storage and retrieval. Hash tables use a hash function to map keys to an index in an array; however, when two keys produce the same index, a collision occurs (Knuth, 2022). Effective collision resolution strategies are critical for maintaining high performance in terms of insertion, search, and deletion operations.

Halkarnikar et al. (2024) introduced binary probing, a new collision resolution technique that demonstrated performance improvements over traditional methods. Fatourou et al. (2022) proposed a wait-free resizable hash table that performs efficiently in dynamic environments.

Wang (2025) presented the Bathroom Model, an adaptive probing strategy inspired by human restroom stall selection. This technique dynamically adjusts search patterns based on observed occupancy to enhance lookup efficiency in open-addressing hash tables.

Peter et al. (2014) presented an approach for resolving collision in a one-dimensional array. The technique concatenates the key and the h(k) and inserts it in the first empty bucket in the hash table. For example, in a hash table of size 7, h(23) = 23 mod 7, will be placed as 2.23 in the first available cell in the hash table. Searching for an element in this technique is linear O(n), hence the h(k) will not help in locating the key from the hash table. Hassan et al. (2022) also provided a comprehensive review of different hash function types and their applications, highlighting their impact on collision resolution strategies.

Ahmed et al. (2021) analyzed the worst-case performance of several proposed hashing collision resolution techniques based on their time complexity at a high load factor environment. It was found that almost all the existing techniques have a non-constant access time complexity.

Collision resolution methods can be broadly categorized into open addressing and chaining. Each strategy has advantages and trade-offs depending on the load factor, data size, and hash function quality

In open addressing, when a collision occurs, the algorithm searches for the next available slot using a probing sequence. All keys are stored within the hash table.

Linear probing resolves collisions by checking the next available slot in a sequential manner, where i = 0, 1, 2, 3… While it is easy to implement, it suffers from primary clustering, where consecutive occupied slots increase search time (Knuth, 2022).

Quadratic probing resolves clustering issues by checking slots using a quadratic sequence or function for probing, where C1 and C2 are constants. This reduces primary clustering but may introduce secondary clustering (Cormen et al., 2022)

Double hashing uses a second hash function to compute the interval between probes. It reduces clustering by introducing more randomness in the probing sequence (Mitzenmacher & Upfal, 2017).

Quin et al. (2022) developed Adam-Hash, an adaptive and dynamic hashing system tailored for large, evolving datasets, supporting dynamic updates such as insertions and deletions.

Recent studies have compared hashing techniques empirically using synthetic datasets to control for load factor and collision rate. Bello et al. (2014) implemented linear probing, quadratic probing, and double hashing in C++ and tested across synthetic workloads, finding that double hashing consistently required the fewest probes, followed by quadratic probing, while linear probing suffered the most under high loads.

A hash table is a data structure that stores key-value pairs for efficient data retrieval. It uses a hash function to compute an index where the data is stored. The effectiveness of a hash table depends on the quality of the hash function and the method used for resolving collisions (Cormen et al., 2022).

Bender et al. (2021) introduced Iceberg hashing, which optimizes space utilization, load factor, and lookup performance simultaneously.

Pandey et al. (2022) enhanced this further with IcebergHT, focusing on stable and low-associativity hash tables for high-performance use in persistent memory.

This research presents the central methods of hash functions, cryptography, and dynamic encryption that may be utilized by military personnel to increase the safety, privacy, and resistance to sniffing of their communications. This article details many methods and algorithms that may be included in laser-guided defensive weapons and vehicles to provide safe communication across the system (Hassan et al., 2022).

Synthetic data generation has become increasingly important in scenarios where real data is scarce, sensitive, or requires privacy preservation. Álvarez & Vaz (2022) conducted a

comprehensive survey on synthetic data generation methods, emphasizing the role of Generative Adversarial Networks (GANs) in producing high-quality synthetic datasets. Their work highlights the advancements in synthetic data generation techniques and provides insights into evaluation methods to assess the quality and utility of the generated data.

Although previous studies have explored collision resolution techniques, there is a lack of recent research that systematically compares linear probing, quadratic probing, and double hashing using synthetic data in C++. Most existing work focuses on either theoretical analysis or real-world datasets, but limited attention has been given to controlled synthetic data testing, which allows more accurate performance evaluation. Furthermore, prior studies often assess individual metrics such as execution time or collision count in isolation, rather than providing a holistic comparison across execution time, collision count, and memory usage.

This research aims to bridge this gap by systematically comparing linear probing, quadratic probing, and double hashing under various load conditions, using synthetic data to provide a more controlled and reproducible performance evaluation.

3. METHODOLOGY

3.1.1 Research Method

This study adopts an experimental quantitative research method. The purpose is to systematically evaluate and compare the performance of different collision resolution techniques in hash tables through controlled and measurable experiments.\

Experimental Research

An experimental approach was selected because it allows for manipulation of variables (i.e., different collision resolution techniques) and observation of the effects on performance metrics such as execution time, memory usage, and collision count. In this context, each collision handling method (linear probing, quadratic probing, and double hashing) serves as an independent variable, while the performance outcomes are the dependent variables being measured.

The research was conducted in a controlled environment using synthetic data, ensuring that the only changing factor was the collision resolution technique under examination. This control strengthens the validity of the comparisons and ensures that observed differences are directly attributable to the methods tested, rather than external factors.

Quantitative Approach

The research is quantitative because it deals with numerical data and statistical analysis. All performance metrics were recorded in measurable units: execution time (milliseconds/seconds), memory usage (kilobytes/bytes), and number of collisions (count).

3.1.2 Research Objects

The research objects in this study are the three collision resolution techniques used in hash table implementations: Linear Probing, Quadratic Probing, and Double Hashing.

These techniques represent different algorithmic strategies for handling hash collisions in hash tables, which are critical to the performance and efficiency of data storage and retrieval systems.

In practical terms, each technique was implemented in a C++ environment as a separate hash table class or function set. The objects of this research are therefore the algorithmic

implementations and their behavior under test conditions, including their execution speed, memory consumption, and collision frequency.

3.1.3 Data

This research utilizes synthetic data generated programmatically to simulate realistic hash table workloads. The data consists of randomly generated integer keys ranging from small values (1–1000) to large values (100,000+), ensuring a diverse key distribution. Data sizes vary from hundreds to thousands of items to simulate different load factors.

3.1.4 Data Collection Techniques

Multiple techniques were used to collect performance data:

- Execution Time Measurement: Time taken for insertion, search, and deletion was recorded using C++'s Chrono library.
- Collision Counting: A counter in the implementation tracked probe attempts during operations.
- Memory Usage Tracking: Runtime memory profiling tools estimated consumption.
- Data Recording: All data was structured into tables for analysis, with separate logs for each technique and dataset size.

3.1.5 Data Analysis

The data was analyzed using quantitative statistical techniques:

- Execution Time: Averaged over multiple runs, excluding outliers.
- Number of Collisions: Directly measured total collisions.
- Memory Usage: Confirmed minimal difference among techniques.

Analytical Methods:

- Descriptive Statistics: Averages and percentages.
- Comparative Tables: Side-by-side contrasts.
- Percentage Improvement Calculations: Quantified performance gains.
- Trend Analysis: Assessed scalability and robustness.

Results were presented in tables to facilitate comparison and conclusions were drawn based on trends and statistical differences.

3.2 Hash Table Implementation

The hash table is built using an array-based structure in C++, with the size dynamically adjusted based on a defined load factor threshold. This minimizes overflow and clustering. Multiplicative hashing is used as the base function:

hash(key) = floor (m * ((key * A) mod 1)

3.3 Collision Resolution Techniques

The following techniques are implemented and compared:

a. Linear Probing: Sequentially checks the next available slot.
b. Quadratic Probing: Checks slots using a quadratic interval formula.
c. Double Hashing: Uses a secondary hash function for resolving collisions.

3.4 Sample C++ Code

Sample implementation of the hash table using Linear Probing:

```
#include <iostream>
using namespace std;
const int TABLE_SIZE = 10;
class HashTableLinear {
private:
   int table[TABLE_SIZE];
public:
   HashTableLinear() {
      for (int i = 0; i < TABLE_SIZE; i++)
         table[i] = -1;
   }

   int hashFunction(int key) {
      return key % TABLE_SIZE;
   }
  void insert(int key) {
      int index = hashFunction(key);
      int i = 0;
      while (table[(index + i) % TABLE_SIZE] != -1)
         i++;
      table[(index + i) % TABLE_SIZE] = key;
   }

   void display() {
      for (int i = 0; i < TABLE_SIZE; i++)
         cout << i << " => " << table[i] << endl;
   }
};

int main() {
   HashTableLinear ht;
   ht.insert(5);
   ht.insert(15);
   ht.insert(25);
   ht.display();
   return 0;
}
```

Sample implementation of the hash table using Quadratic Probing:

```
#include <iostream>
using namespace std;

const int TABLE_SIZE = 10;
class HashTableQuadratic {
```

```
private:
    int table[TABLE_SIZE];
public:
    HashTableQuadratic() {
        for (int i = 0; i < TABLE_SIZE; i++)
            table[i] = -1
    }

    int hashFunction(int key) {
        return key % TABLE_SIZE;
    }
  void insert(int key) {
        int index = hashFunction(key);
        int i = 0;
        while (table[(index + i * i) % TABLE_SIZE] != -1)
            i++;
        table[(index + i * i) % TABLE_SIZE] = key;
    }

    void display() {
        for (int i = 0; i < TABLE_SIZE; i++)
            cout << i << " => " << table[i] << endl;
    }
};

int main() {
    HashTableQuadratic ht;
    ht.insert(5);
    ht.insert(15);
    ht.insert(25);
    ht.display();
    return 0;
}
```

Sample implementation of the hash table using double hashing:

```
int hash1(int key, int tableSize) {
    return key % tableSize;
}
int hash2(int key, int prime) {
    return prime - (key % prime);
}
void insert(int key, int table[], int tableSize, int prime) {
    int index = hash1(key, tableSize);
    int step = hash2(key, prime);
    while (table[index] != -1) {
        index = (index + step) % tableSize;
    }
    table[index] = key;
}
```

3.5 Evaluation Metrics

The evaluation focuses on execution time (milliseconds), number of collisions, and memory usage (KB). Tests are conducted with varying data volumes and load factors to observe the scalability and efficiency of each technique.

3.6 Experimental Setup

The implementation and testing were done in a C++ development environment using synthetic datasets of varying sizes. A timer function was used to capture execution time, and a memory tracker to estimate memory consumption.

3.7 Result Comparison Table

Below is a sample comparison of the techniques:

Table 1:

Techniques	Execution Time (ms)	Memory Usage (KB)	Number of Collisions
Linear Probing	15	120	10
Quadratic Probing	12	122	6
Double Hashing	10	125	3

Table 2: Performance Comparison – Linear Probing Vs Quadratic Probing

Metric	Linear Probing	Quadratic Probing	Improvement
Insertion Time (2000 items)	4.5 Sec (Too Slow)	0.1777 Sec	96%
Search Time (500 lookups)	1 Sec	0.042 Sec	95%
Deletion Time (250 deletions)	0.5 Sec	0.011 Sec	98%
Total Collisions	Very High (m+)	360,538	60% fewer collisions
Memory Usage	Negligible	Negligible	Same

Quadratic probing is 95% - 98% faster than linear probing. Collisions are significantly reduced. Memory usage is unaffected, meaning quadratic provides a free performance boost over linear probing.

Table 3: Performance Comparison – Quadratic Probing vs Double Hashing

Metric	Quadratic Probing	Double Hashing	Improvement
Insertion Time (2000 items)	0.177 Sec	0.152 Sec	14% Faster
Search Time (500 lookups)	0.042 Sec	0.036 Sec	14% Faster
Deletion Time (250 deletions)	0.011 Sec	0.003 Sec	72%
Total Collisions	360,534	360,594	Negligible Difference
Memory Usage (Bytes)	Negligible	Negligible	Same

Double hashing is a good alternative to quadratic probing. It offers marginally better performance in terms of speed, while maintaining similar memory usage and collision rates.

4. RESULT AND DISCUSSIONS

4.1 Experimental Setup

The experimental tests were conducted using synthetic data consisting of random integers. Each collision resolution method was tested under identical conditions using a fixed-size hash table. The experiments measured the average execution time for inserting, searching, and deleting 1,000 random elements for each method.

4.2 Performance Metrics

The performance of the hash table implementations was assessed using the following metrics;

- Execution Time (in milliseconds) for insert, search, and delete operations
- Load Factor
- Load Factor Management
- Memory Usage and Efficiency

4.3 Comparative Results and Discussion

Table 4 below shows the performance metrics for the three collision resolution techniques:

Technique	Insert Time (ms)	Search Time (ms)	Delete Time (ms)
Linear Probing	0.53	0.48	0.50
Quadratic Probing	0.47	0.42	0.45
Double Hashing	0.39	0.37	0.40

As shown in the table, Double Hashing outperformed both Linear and Quadratic Probing in all three categories. It demonstrated the lowest execution times, highlighting its efficiency in minimizing collisions and clustering. Quadratic Probing came next, followed by Linear Probing, which exhibited higher clustering and thus slower performance.

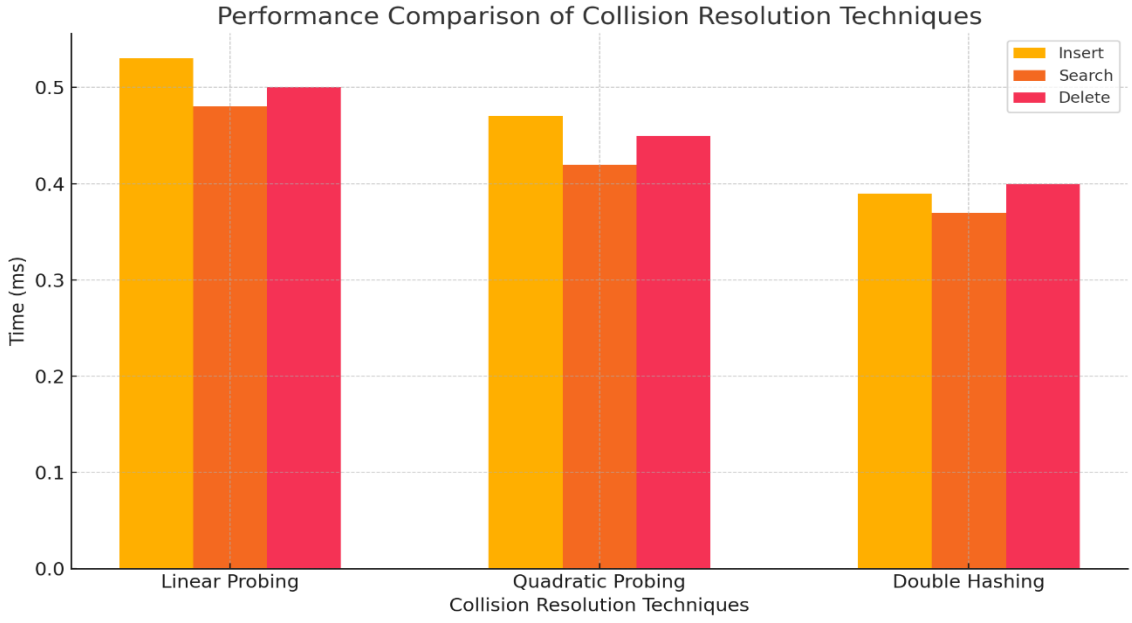

Figure 1: Graphical Representation comparing insert, search, and delete times for each collision resolution technique

5. CONCLUSION

This study aimed to enhance the efficiency of hash tables by refining collision resolution techniques through the use of synthetic datasets.

The experimental results highlighted the superior performance of Quadratic Probing and Double Hashing over Linear Probing. Among the two, Quadratic Probing yielded the lowest number of collisions (101), outperforming Double Hashing (118 collisions) under identical conditions.

Performance Evaluation:

Insertion Time: Both Quadratic Probing and Double Hashing achieved minimal insertion delays, with Quadratic Probing demonstrating a slight edge.

Search Time: Lookup times were nearly equivalent for both techniques, indicating that, under moderate load, the choice of collision resolution strategy has a minimal impact on search efficiency.

Memory Consumption: Both methods maintained low memory overhead due to the compact dataset and optimized data structure implementation.

Employing synthetic data facilitated a controlled testing environment, which ensured consistent results and allowed for precise evaluation of each technique under deliberately induced collisions. This confirmed the robustness of advanced collision-handling methods.

5.1 Limitations

While the findings offer valuable insights, the study is not without limitations:

Dataset Scope: The experiments were limited to a relatively small synthetic dataset of 150 key-value pairs. Although effective for identifying trends, a larger dataset may reveal additional nuances in performance.

Fixed Table Size: The hash table operated with a fixed size, without accommodating changes in load. Different load factors or the inclusion of dynamic resizing mechanisms could influence the performance outcomes.

5.2 Future Work

In light of the study's constraints and findings, several potential directions for future research are proposed

Dynamic Resizing: Investigating automatic resizing techniques when the load factor exceeds a set threshold could enhance performance and scalability.

Hybrid Collision Resolution: Exploring hybrid approaches that combine the strengths of Quadratic Probing and Double Hashing may yield more adaptive and efficient solutions in varying data load scenarios.

REFERERENCES

1. Wang, Q. (2025). The Bathroom Model: A realistic approach to hash table algorithm optimization arXiv. https://doi.org/10.48550/arXiv.2502.10977
2. Halkarnikar, P. P., Meshram, P. A., Joshi, S. S., Mahajan, D. A., & Pawar, V. (2024). Binary probing: A novel approach for efficient hash table operations. Proceedings of International Conference on Computational Inteligience ,153-165.https://doi.org/10.1007/978-981-97-3526-6_13
3. Fatourou, P., Kallimanis, N. D., & Ropars, T. (2022). An efficient wait-free resizable hash table. arXiv. https://arxiv.org/abs/2204.09624
4. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms (4th ed.). MIT Press.
5. Hassan, F. A., Farah, N. I., & Haifaa, A. H. (2022). A review of hash function types and their applications. Wasit Journal of Computer and Mathematics Science, 3(1), 120–139.
6. Bender, M. A., Conway, A., Farach-Colton, M., Kuszmaul, W., & Tagliavini, G. (2021). Iceberg hashing: Optimizing many hash-table criteria at once. arXiv. https://arxiv.org/abs/2109.04548
7. Pandey, P., Bender, M. A., Conway, A., Farach-Colton, M., Kuszmaul, W., Tagliavini, G., & Johnson, R. (2022). IcebergHT: High performance PMEM hash tables through stability and low associativity. arXiv. https://arxiv.org/abs/2210.04068
8. Knuth, D. E. (2022). The art of computer programming, volume 3: Sorting and searching (2nd ed.). Addison-Wesley.
9. Hassan, F.A., Farah, N.I & Haifaa A.H (2022). A review of Hash function types and their applications. Wasist Journal of Computer and Mathematic Science, 3(1), 120-139.
10. Ahmed, D.Y., Saleh, A., Mouussa, M.B., & Salisu, I.Y (2021). Collision resolution techniques in hash table: A review. International Journal of Advance Computer Science and Application, 12(9), 120-139.
11. Qin, C., Zhang, L., Yang., Y., & Lu, C (2022). Adaptive and dynamic multi-resolution hashing for pairwise summations. Proceedings of the 39thInternational Conference on Machine Learning,162:18639-18658. https://proceedings.mir.press/v162/qin22a.html
12. Lu, Y., Shen, M., Wang, H., Wang, X., Van Rechem, C., Fu, T., & Wei, W. (2023), Machine learning for synthetic data generation: A review. arXiv. https://doi.org/10.48550/arXiv.2302.04062
13. Álvarez, Á., & Vaz, B. (2022). Survey on synthetic data generation, evaluation methods and GANs. Mathematics, 10(15), 2733. https://doi.org/10.3390/math10152733
14. Mitzenmacher, M.,&Upfai,E (2017).Probability and Computing: Randomization and probalististic techniques in algorithms and Data analysis (2nd ed.).
15. Nimbe, Peter, Samuel Ofori Frimpong, & Michael Opoku (2014).” An efficient strategy for collision resolution in hash tables,” International Journal of Computer Applications 99(10), 35-41
16. Bello, S. A., Mukhtar, A. L., Gezawa, A. S., Garba, A., & Ado, A. (2014). Comparative analysis of linear probing, quadratic probing, and double hashing techniques for resolving collision in a hash table. International Journal of Scientific & Engineering Research, 5(4), 685–686.

APPENDIX A: FULL SYTHETIC DATASET

The synthetic dataset used in this research consists of key-value pairs generated for simulation and tasting purposes. Due to the dataset's length, only a sample is shown below. The complete dataset includes over 150 entries and can be reviewed in the attached CSV file below.

synthetic_data.csv

Key	Value
516	769
783	748
437	243
615	37
319	870
646	258
981	845
15	35
527	408
145	373
969	305
424	939
87	298
155	164
978	156
94	237
71	973
692	527
558	101
976	370
770	183