Hash Function Analysis Report for Translator Program

Submitted By: Meera Alzaabi

Instructor: Khalid Mengal

Professor: Talal Rahwan

Course Code: CS-UH 1050

Course: Data Structures

Table of Contents

1 Introduction

A hash function is a mathematical algorithm that modifies the input data like string or number into a fixed size typically a hash-code. The output is utilized for fast data lookup, comparison and integrity checks.[1] The hash functions are significant cryptographic primitive. They construct a message's digest, which is a small, fixed-length bitstring. The message digest, also known as the hash value, can be thought of as a message's fingerprint, or a distinctive copy of a message. [2]In this report, an in-depth analysis of the hash functions used in the program, a C++ dictionary management system employing a hash table with open-addressing and linear probing is presented. The analysis focuses on the original polynomial hash function and compares its performance against two other hash functions, FNV-1a and MurmurHash function. Using real data from German, Spanish, and French datasets, this report documents the design, implementation, and experimental results of the translator.

1.1 Types of Hash Functions

- The Polynomial Hash Function uses a mathematical polynomial representation of the input to generate a hash value, often used for string matching algorithms like Rabin-Karp.[3]
- FNV-1a is a non-cryptographic, efficient hash function widely used in hash tables and checksum applications. It works by XORing the input with a base value and multiplying it by a constant prime number. [4]
- MurmurHash, known for its speed and good distribution properties, is favored in high-performance applications, especially large datasets and distributed systems. [5]

1.2 Features

Polynomial Hash*:* The code is concise (hash = (hash * 31 + ch) % capacity) and requires no complex operations or constants, making it ideal for quick development and maintenance. [3]

FNV-1a*:* Requires defining specific constants (e.g., 2166136261 offset, 16777619 prime) and an XOR operation, adding slight complexity. [4]

MurmurHash3*:* Involves multiple rounds of bit manipulation (rotations, multiplications, final mixing), making it the most complex and hardest to implement without errors (as seen in the initial compilation issues).[5]

1.3 Experimental Results and Discussion

The existing hash function in the program uses a polynomial hashing algorithm with a base of 31. It converts the input word to lowercase for case-insensitive comparison and computes the hash iteratively:

$$\text{hash} = (\text{hash} \times 31 + \text{char})$$

If the input word is empty, it returns 0. This method is simple and leverages the ASCII values of characters to generate the hash.

1.3.1 *Theoretical Properties*

- The polynomial approach with base 31 provides moderate distribution but can suffer from clustering due to its linear nature, especially with similar strings (e.g., ”cat” and ”bat”).
- The time complexity is $O(n)$, where n is the length of the string.

1.3.2 *Strengths*

- Simple to implement and computationally inexpensive.
- Works well for small datasets with low collision rates.[3]

1.3.3 *Weaknesses*

- Poor distribution of large datasets, leading to a higher collision rate
- Sensitive to the choice of base and capacity, which can exacerbate clustering in linear probing.

1.3.4 *Results*

Table 1 Polynomial Results

Dictionary	Total No. of Collisions	Avg. No. of Collisions	Size of Hashtable
German	936574	1.15	812739
Spanish	430	0.01	31278
French	5824	0.05	109698

Table 2 FNV-1a Results

Dictionary	**Total No. of Collisions**	**Avg. No. of Collisions**	**Size of Hashtable**
German	1107923	1.31	845692
Spanish	479	0.02	31278
French	7070	0.05	119189

Table 3 Murmur Hash Results

Dictionary	Total No. of Collisions	Avg. No. of Collisions	Size of Hashtable
German	1109095	1.31	845692
Spanish	424	0.01	31278
French	7052	0.06	119189

2 Design and Implementation

2.1 Polynomial Hash Function

2.2 Pseudo Code

The Polynomial hash function computes a hash value by treating the string as a polynomial with a base of 31, accumulating the character values, and applying a modulus to fit the table capacity.

```
FUNCTION hashCode(word):
    hash  0
    lowerWord  toLower(word)
    IF lowerWord is empty THEN
        RETURN 0
    END IF
    FOR each character ch in lowerWord:
        hash  (hash * 31 + ch) mod capacity
    END FOR
    RETURN hash
END FUNCTION
```

Figure 1 Pseudo Code for Polynomial hash function

2.3 Implementation of C++ for Polynomial

The following is the C++ implementation of the Polynomial hash function as used in HashTable.

```cpp
unsigned long HashTable::hashCode(const std::string& word) const {
    unsigned long hash = 0; // Initialize hash value to 0
    std::string lowerWord = toLower(word); // Convert word to lowercase for consistency
    if (lowerWord.empty()) { // Check if the word is empty
        return 0; // Return 0 for empty input
    }
    for (const auto& ch : lowerWord) { // Iterate over each character in the lowercase word
        hash = (hash * 31 + ch) % capacity; // Update hash using polynomial method: hash = (hash * 31 + char) mod capacity
    }
    return hash; // Return the final hash value
}
```

2.4 FNV-1a hash function

The FNV-1a hash function uses a combination of XOR and multiplication with a prime number to mix the bits of the input string, followed by a modulus operation to fit the table capacity.

```
FUNCTION hashCode(word):
    FNV_PRIME 16777619
    FNV_OFFSET  2166136261
    hash  FNV_OFFSET
    lowerWord  toLower(word)
    IF lowerWord is empty THEN
        RETURN 0
    END IF
    FOR each character ch in lowerWord:
        hash  hash XOR ch
        hash  hash * FNV_PRIME
    END FOR
    hash  hash mod capacity
    RETURN hash
END FUNCTION
```

Figure 2 Pseudo Code

The following is the C++ implementation of the FNV-1a hash function.

```
// Define a hash table class that maps a string and word to a hash value
unsigned long HashTable::hash(const std::string &word) const {
    // Initialize constants for hashing
    const unsigned long FNV_PRIME = 16777619u;  // FNV prime number for hashing
    const unsigned long FNV_OFFSET = 2166136261u;  // FNV offset basis for hashing
    unsigned long hash = FNV_OFFSET;  // Start with the FNV offset as the initial hash value

    // Check if the word is empty; if so, return 0 as the hash
    if (lowerWord.empty()) {
        return 0;
    }

    // Loop through each character in the word to compute the hash
    for (const auto &ch : lowerWord) {
        hash ^= static_cast<unsigned long>(ch);  // XOR the hash with the character's value
        hash *= FNV_PRIME;  // Multiply the hash by the FNV prime to mix the bits
    }

    return hash % capacity;  // Return the final hash value, modulo the table's capacity
}
```

2.5 MurmurHash

MurmurHash processes the input string in 4-byte blocks, mixing each block with constants, rotations, and multiplications, followed by handling remaining bytes (tail) and a final mixing step to ensure good distribution.

```
FUNCTION hashCode(word):
    c1  0xcc9e2d51
    c2  0x1b873593
    seed  0
    hash  seed
    lowerWord  toLower(word)

  IF lowerWord is empty THEN
      RETURN 0
  END IF
  data  convert lowerWord to byte array
  len  length of lowerWord
  nblocks  len / 4
  FOR i  0 to nblocks-1:
      k  read 4 bytes from data at position i*4
      k  k * c1
      k  rotateLeft(k, 15)
      k  k * c2
      hash  hash XOR k
      hash  rotateLeft(hash, 13)
      hash  hash * 5 + 0xe6546b64
  END FOR
  tail  remaining bytes (len mod 4)
  k  0
  IF tail has 3 bytes THEN
      k  k XOR (tail[2] << 16)
  END IF
  IF tail has 2 or more bytes THEN
      k  k XOR (tail[1] << 8)
  END IF
  IF tail has 1 or more bytes THEN
      k  k XOR tail[0]
      k  k * c1
      k  rotateLeft(k, 15)
      k  k * c2
      hash  hash XOR k
  END IF
  hash  hash XOR len
  hash  finalMix(hash) % Using fmix32: hash ^= hash >> 16; hash *=
     0x85ebca6b; etc.
  hash  hash mod capacity
  RETURN hash
END FUNCTION
```

Figure 3 Pseudo Code

Implementation of C++

The following is the C++ implementation of the MurmurHash3 hash function.

```
class HashTable {
private:
    // Helper function for MurmurHash3: Rotate left
    // Rotates the bits of x by r positions to the left
    inline uint32_t rotl32(uint32_t x, int8_t r) const {
        return (x << r) | (x >> (32 - r));
    }
    // Helper function for MurmurHash3: Final mix
    // Mixes the bits of h to ensure good distribution
    inline uint32_t fmix32(uint32_t h) const {
        h ^= h >> 16;  // XOR with right-shifted h to mix bits
        h *= 0x85ebca6b;  // Multiply by a constant to further mix
        h ^= h >> 13;  // XOR with another right-shifted h
        h *= 0xc2b2ae35;  // Multiply by another constant
        h ^= h >> 16;  // Final XOR for bit mixing
        return h;  // Return the mixed hash
    }
public:
    unsigned long hashCode(const std::string& word) const {
        std::string lowerWord = tolower(word);  // Convert the input word to lowercase
        if (lowerWord.empty()) {  // Check if the word is empty
            return 0;  // Return 0 for empty strings
        }
        // Initialize constants for MurmurHash3
        const uint32_t c1 = 0xcc9e2d51;  // Constant for bit mixing
        const uint32_t c2 = 0x1b873593;  // Another constant for bit mixing
        const uint32_t seed = 0;  // Seed value for hashing

        const char* data = lowerWord.c_str();  // Get the character array of the word
        size_t len = lowerWord.length();  // Get the length of the word
        const int nblocks = len / 4;  // Number of 4-byte blocks in the word

        // Initialize the hash with the seed value
        uint32_t h1 = seed;

        // Process the word in 4-byte blocks
        const uint32_t* blocks = reinterpret_cast<const uint32_t*>(data + nblocks * 4);
        for (int i = -nblocks; i; i++) {  // Iterate over each block
            uint32_t k1 = blocks[i];  // Get the current 4-byte block
            k1 *= c1;  // Multiply by c1 for bit mixing
            k1 = rotl32(k1, 15);  // Rotate left by 15 bits
            k1 *= c2;  // Multiply by c2 for further mixing
            h1 ^= k1;  // XOR with the current hash
            h1 = rotl32(h1, 13);  // Rotate the hash left by 13 bits
            h1 = h1 * 5 + 0xe6546b64;  // Mix with a constant
        }
        // Handle the remaining bytes (tail) after the last 4-byte block
        const uint8_t* tail = reinterpret_cast<const uint8_t*>(data + nblocks * 4);
        uint32_t k1 = 0;  // Initialize k1 for the tail
        switch (len & 3) {  // Process remaining bytes based on length modulo 4
            case 3: k1 ^= tail[2] << 16;  // Handle the third byte
            case 2: k1 ^= tail[1] << 8;   // Handle the second byte
            case 1: k1 ^= tail[0];        // Handle the first byte
                    k1 *= c1;  // Multiply by c1
                    k1 = rotl32(k1, 15);  // Rotate left by 15
                    k1 *= c2;  // Multiply by c2
                    h1 ^= k1;  // XOR with the hash
        }

        h1 ^= len;  // XOR the hash with the length of the word
        h1 = fmix32(h1);  // Final mix to ensure good distribution

        return h1 % capacity;  // Return the hash value modulo the table's capacity
    }
};
```

3 Summary

The polynomial hash function performs well for smaller datasets but suffers with larger datasets. The polynomial hash function is a popular choice due to its ease of implementation. The FNV-1a and MurmurHash functions provide considerable enhancements, with the latter being particularly useful in large-scale applications.

4 References

[1] C. Paar, J. Pelzl, C. Paar, J. P.-: A. T. for S. and Practitioners, and undefined 2010, "Hash functions," *Springer*, pp. 293–317, 2010, doi: 10.1007/978-3-642-04101-3_11.

[2] A. Maetouq *et al.*, "An overview of cryptographic hash functions and their uses," *target0.be*, vol. 9, no. 8, 2018, Accessed: May 04, 2025. [Online]. Available: http://target0.be/madchat/crypto/papers/paper879.pdf

[3] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger, "Polynomial hash functions are reliable," *Springer*, vol. 623 LNCS, pp. 235–246, 1992, doi: 10.1007/3-540-55719-9_77.

[4] C. Hayes, D. M.-2024 35th I. S. and Systems, and undefined 2024, "An Evaluation of FNV Non-Cryptographic Hash Functions," *ieeexplore.ieee.org*, Accessed: May 05, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10603139/

[5] F. Yamaguchi, H. N.-2013 19th I. International, and undefined 2013, "Hardware-based hash functions for network applications," *ieeexplore.ieee.org*, Accessed: May 05, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6781990/