# Hash Table Implementation Report: Comparative Analysis of Collision Resolution Techniques

**Table of Contents**

Yashwant Chandrakant Bhosale
612303039
SY Comp. Div 1

# Introduction

1. **What is Hashing?**
   Hashing is a technique in searching used to efficiently store and retrieve data. At its core, hashing is a method of mapping data of arbitrary size to fixed size values using a hash function.
   It is like a sophisticated record table that allows us to access record in near constant time.
2. **How hashing works?**
   Here, instead of searching through entire set of records to find the required record we use some hashing function that magically takes us closer to the record if not exact position of record. Hash function always returns unique hash value for unique key.
3. **Key Components of Hashing:**
   **Hash Function:** A mathematical algorithm that converts input data into a fixed size string of characters.
   **Hash Table:** An array that stores the data using the hash function's output as an index.
   **Key:** The original input data being hashed
   **Hash Value:** The transformed output produced by the hash function.
4. **Basic Hashing Process:**
   1. Get the key from the record
   2. Pass it through the hash function
   3. Generate a unique hash value
   4. Store data at the index corresponding to the hash value
   5. When we want to retrieve data again, we again hash the key of the record and reach the index to retrieve the data.
5. **Important Features**
   **1. Fast Data Retrieval**: O(1) time complexity for most operations
   **2. Efficient Data Storage**: Compact representation of complex data
   **3. Uses:** Used in security applications like password storage.

# 1. Hash Functions

Simple hash function implementation using division method and multiplication method.

## Division Method Hash Function

This method makes use of the property of modulo or reminder which ensures that the resulting hash value is within the range of 0 and size of hash function that is value ranges from 0 to size-1.
**Key Modulo Size**: The key is divided by the size of the hash table, and the remainder of that division is used as the index in the hash table. This ensures that the result is always within the valid range of indices for the hash table (i.e., between `0` and `size-1`).
**Prime Table Size**: For better performance, it's often recommended that the table size (`size`) be a **prime number**. This helps to reduce collisions, as a prime number size tends to distribute keys more evenly across the table.
**Collision Handling**: If two keys produce the same hash value (i.e., a collision), you’ll need to handle collisions, typically using methods like **chaining** (storing collided keys in a linked list at the same table index) or **open addressing**.

This is how a division hash function looks like:

```
int division_hash(int key, int size) {
    return key % size;
}
```

## Multiplication Method Hash Function

The **multiplication hash function** is a method used to generate hash values by multiplying the key with a constant factor, extracting the fractional part of the result, and then scaling it to the size of the hash table.
1. Multiply the key by a constant A (typically the fractional part of the golden ratio).
2. Extract the fractional part of the result.
3. Multiply the fractional part by the table size and take the integer part to obtain the hash value.

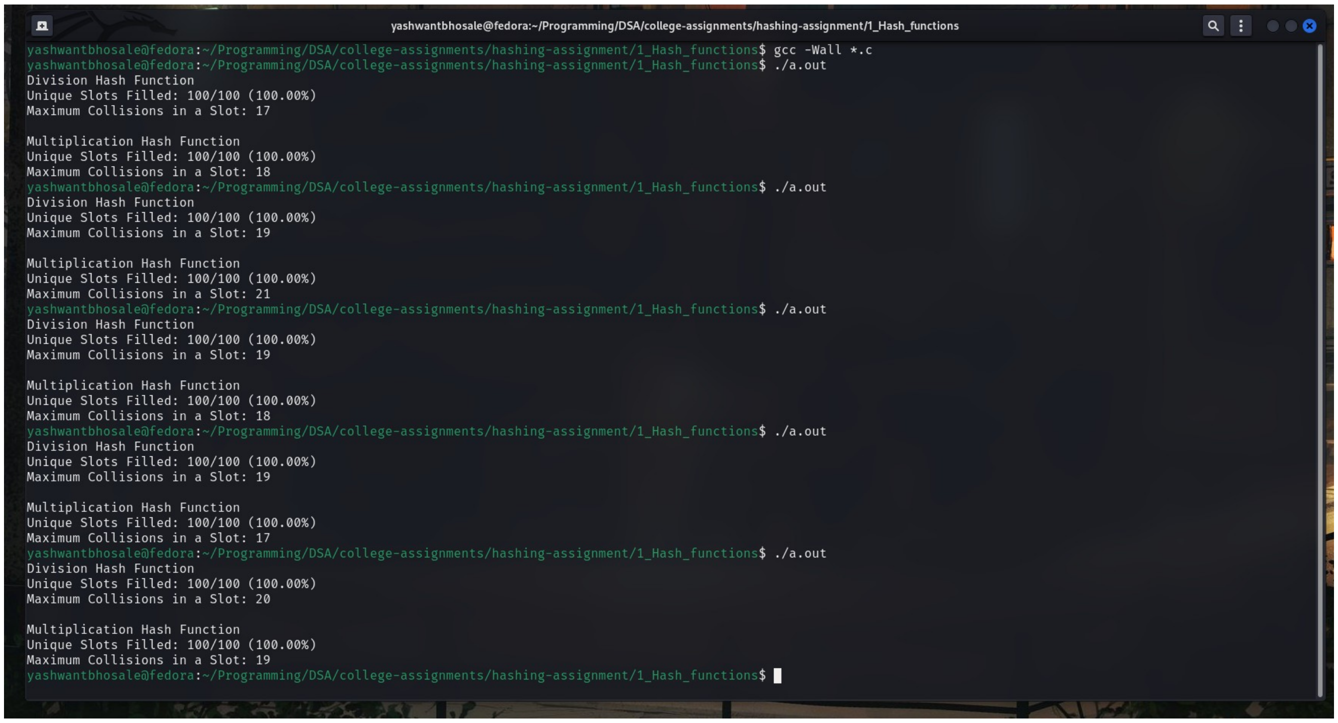
This is how a typical multiplication hash function looks like:

```
int multiplication_hash(int key, int size) {
    double A = 0.6180339887;
    double frac = key * A - (int)(key * A);
    return (int)(size * frac);
}
```

**Results**

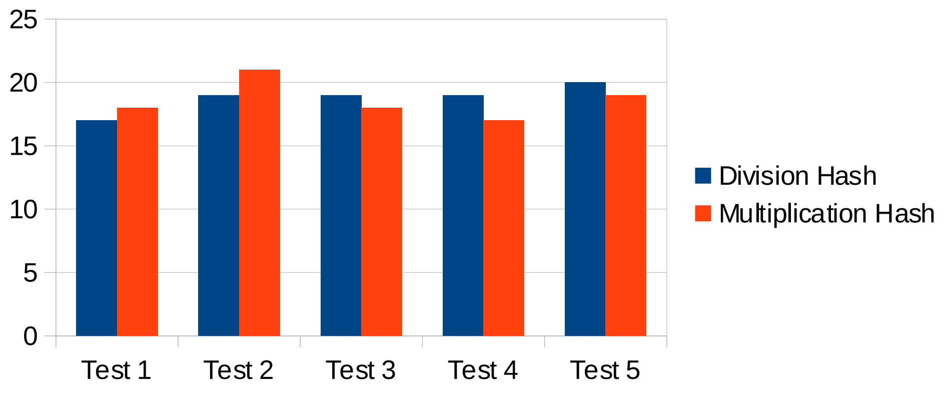
1. **Performance**:
   The maximum collisions for both hashing methods are relatively close in most tests.
2. **Consistency**
   The division hash methods seems to produce similar results across different tests, indicating a more consistent distribution of keys across slots.
   The multiplication hash method shows slightly more variability, suggesting that it might occasionally cluster more keys into a single slot.

   In general there is not a huge difference between performance of both the hashing methods and both perform almost same on variety of random data.



```
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/1_Hash_functions$ gcc -Wall *.c
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/1_Hash_functions$ ./a.out
Division Hash Function
Unique Slots Filled: 100/100 (100.00%)
Maximum Collisions in a Slot: 17

Multiplication Hash Function
Unique Slots Filled: 100/100 (100.00%)
Maximum Collisions in a Slot: 18
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/1_Hash_functions$ ./a.out
Division Hash Function
Unique Slots Filled: 100/100 (100.00%)
Maximum Collisions in a Slot: 19

Multiplication Hash Function
Unique Slots Filled: 100/100 (100.00%)
Maximum Collisions in a Slot: 21
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/1_Hash_functions$ ./a.out
Division Hash Function
Unique Slots Filled: 100/100 (100.00%)
Maximum Collisions in a Slot: 19

Multiplication Hash Function
Unique Slots Filled: 100/100 (100.00%)
Maximum Collisions in a Slot: 18
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/1_Hash_functions$ ./a.out
Division Hash Function
Unique Slots Filled: 100/100 (100.00%)
Maximum Collisions in a Slot: 19

Multiplication Hash Function
Unique Slots Filled: 100/100 (100.00%)
Maximum Collisions in a Slot: 17
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/1_Hash_functions$ ./a.out
Division Hash Function
Unique Slots Filled: 100/100 (100.00%)
Maximum Collisions in a Slot: 20

Multiplication Hash Function
Unique Slots Filled: 100/100 (100.00%)
Maximum Collisions in a Slot: 19
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/1_Hash_functions$ 
```

| **Test** | **Maximum Collisions in a slot** | |
|---|---|---|
| | **Division Hash** | **Multiplication Hash** |
| Test 1 | 17 | 18 |
| Test 2 | 19 | 21 |
| Test 3 | 19 | 18 |
| Test 4 | 19 | 17 |
| Test 5 | 20 | 19 |



**Driver code:**

**hash_table.h**
```
// Hash table functions
#include <stdio.h>
#include <stdlib.h>

/*
    EXPERIMENT:
    Implement a simple hash function using the division method.
    Write a small program that hashes integers using your function,
then observe
    how data is distributed across an array of a fixed size.
*/

typedef struct {
    int key;
    int value;
} Entry;

typedef struct {
    Entry **table;
    int size;
} HashTable;

void init_table(HashTable *ht, int size);
void insert(HashTable *ht, int key, int value);
void delete(HashTable *ht, int key);
int search(HashTable *ht, int key);
void print_table(HashTable *ht);
void free_table(HashTable *ht);
int multiplication_hash(int key, int size);
int division_hash(int key, int size);
```

**hash_table.c**
```
#include <stdio.h>
#include <stdlib.h>
#include "hash_table.h"

int division_hash(int key, int size) {
    return key % size;
}

int multiplication_hash(int key, int size) {
```

```c
    double A = 0.6180339887;
    double frac = key * A - (int)(key * A);
    return (int)(size * frac);
}

void init_table(HashTable *ht, int size) {
    ht->size = size;
    ht->table = (Entry **)calloc(size, sizeof(Entry *));

    return;
}

void insert(HashTable *ht, int key, int value) {
    int index = division_hash(key, ht->size);

    Entry *new_entry = (Entry *)malloc(sizeof(Entry));
    new_entry->key = key;
    new_entry->value = value;

    if(ht->table[index] != NULL) {
        printf("Collision detected at index %d\n", index);
        free(ht->table[index]);
    }

    ht->table[index] = new_entry;

    return;
}

void delete(HashTable *ht, int key) {
    int index = division_hash(key, ht->size);

    free(ht->table[index]);
    ht->table[index] = NULL;

    return;
}

int search(HashTable *ht, int key) {
    int index = division_hash(key, ht->size);

    if (ht->table[index] == NULL) {
        return -1;
    }

    return ht->table[index]->value;
```

```
}

void print_table(HashTable *ht) {
    for (int i = 0; i < ht->size; i++) {
        if (ht->table[i] != NULL) {
            printf("{ Key: %d, Value: %d }\n", ht->table[i]->key,
ht->table[i]->value);
        }else {
            printf("{ ___ }\n");
        }
    }
    return;
}
```

**main.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "hash_table.h"

#define TABLE_SIZE 100
#define NUM_TESTS 1000


void test_hash_functions() {
    int data_mul[TABLE_SIZE] = {0};
    int data_div[TABLE_SIZE] = {0};
    int unique_slots = 0;
    int max_collisions_div = 0;
    int max_collisions_mul = 0;

    for (int i = 0; i < NUM_TESTS; i++) {
        int key = rand() % 10000;
        int index_div = division_hash(key, TABLE_SIZE);
        int index_mul = multiplication_hash(key, TABLE_SIZE);
        data_div[index_div]++;
        data_mul[index_mul]++;
    }

    for (int i = 0; i < TABLE_SIZE; i++) {
        if (data_div[i] > 0) unique_slots++;
        if (data_div[i] > max_collisions_div) max_collisions_div =
data_div[i];
        if (data_mul[i] > max_collisions_mul) max_collisions_mul =
data_mul[i];
    }
    printf("Division Hash Function\n");
    printf("Unique Slots Filled: %d/%d (%.2f%%)\n", unique_slots,
TABLE_SIZE, (float)unique_slots/TABLE_SIZE * 100);
    printf("Maximum Collisions in a Slot: %d\n",
max_collisions_div);

    printf("\nMultiplication Hash Function\n");
    printf("Unique Slots Filled: %d/%d (%.2f%%)\n", unique_slots,
TABLE_SIZE, (float)unique_slots/TABLE_SIZE * 100);
    printf("Maximum Collisions in a Slot: %d\n",
max_collisions_mul);

    return;
}

int main() {
    srand(time(NULL));

    test_hash_functions();
    return 0;
}
```

## 2. Collision Handling using Chaining

### What is Collision?

Since a hash function gets us a small number for a key which is a big integer or string, there is a possibility that two keys result in the same value. The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique.

### What is Chaining?

Chaining is a technique used in hash tables to handle collisions (when multiple keys map to the same index). Instead of overwriting existing data, chaining stores all elements that hash to the same index in a linked list (or another data structure) at that index.

### How does it work?

**1. Hashing**: A hash function computes the index for a given key.
**2. Collision Handling**: If a collision occurs (i.e., another key is already stored at that index), the new key-value pair is added to the linked list at that index.
**3. Insertion**: New elements are appended to the end of the list at the hashed index.
**4. Search**: The algorithm traverses the list at the index to find the key.
**5. Deletion**: The list at the index is traversed, and the matching key-value pair is removed.

Typical insert function in chaining based hash table:

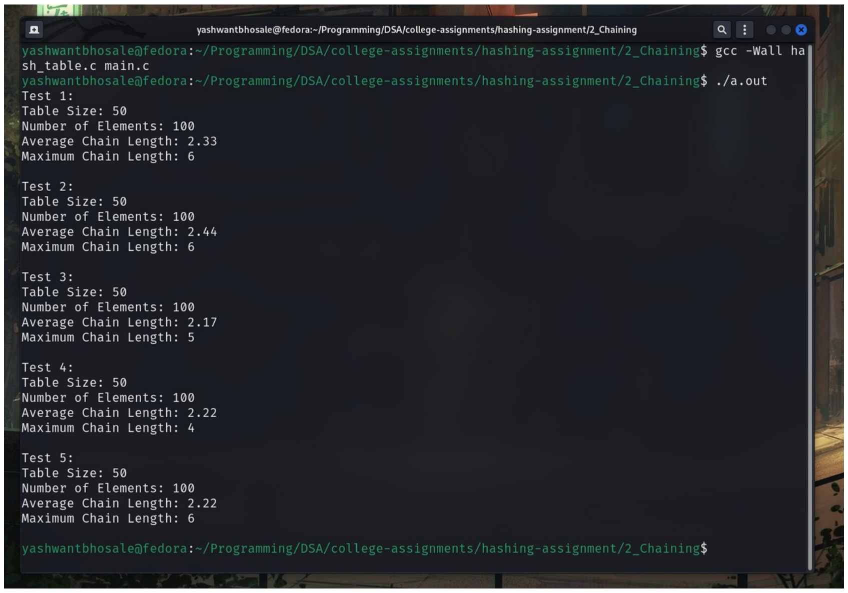
```
void insert(HashTable *ht, int key, int value) {
    int index = hash(key, ht->size);
    Entry *new_entry = (Entry *)malloc(sizeof(Entry));
    new_entry->key = key;
    new_entry->value = value;
    new_entry->next = NULL;

    if(ht->table[index] == NULL) {
        ht->table[index] = new_entry;
    }else {
        Entry *cursor = ht->table[index];
        while(cursor->next != NULL) {
            cursor = cursor->next;
        }
        cursor->next = new_entry;
    }
    return;
}
```

**Results:**

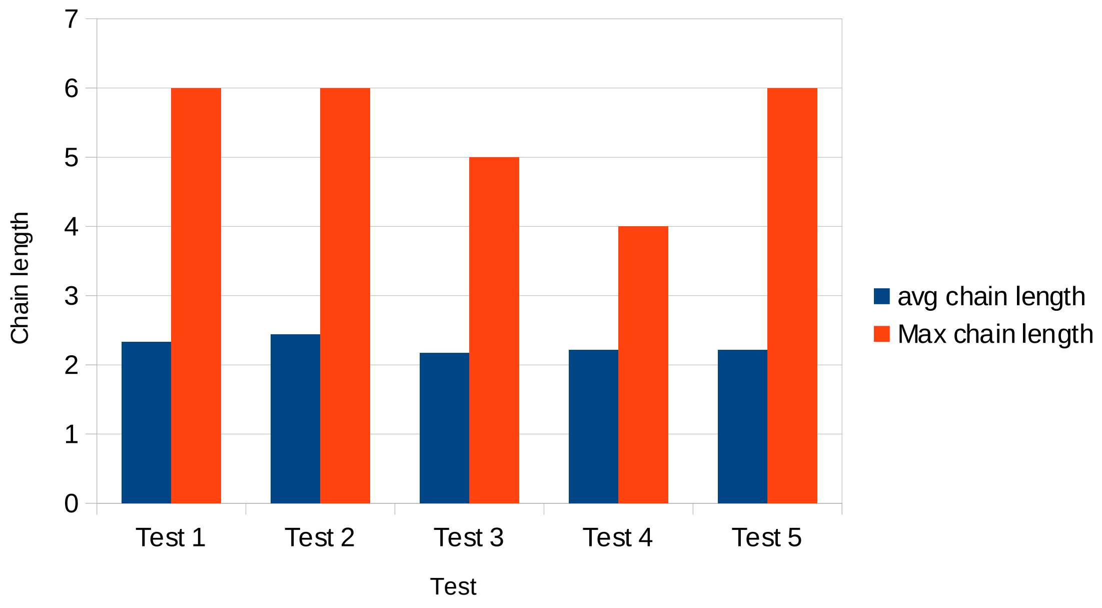
1. **Average Chain length**
   For a given table size and number of elements the that is given load factor the chain length almost stays constant across random set of tests.
   It is approximately equal to the load factor which is number of elements over table size.
2. **Max chain length**
   When we try to fit larger number of elements in smaller table size we get larger chain lengths and as we increase the table size the keeping the number of elements constant the maximum chain length decreases.
   Minimum chain length is optimal for faster access times .
3. **Performance Implications**
   <u>Shorter chains:</u> result in faster average search times since fewer elements need to be checked at each index
   <u>Longer chains:</u> Increase search and insert times for keys in those chains especially as the load factor approaches or exceeds 1.
4. **Load Factor Impact**
   At <u>low load factor</u> (which implies larger empty space in the table), chains tend to remain short and performance is optimal.
   At <u>high load factor</u> (which implies lesser to no empty space in the table), chains tend to grow longer leading to degraded performance.

**1) Table size = 50, Number of elements = 100**

| Tests | No. of elements | Avg chain length | Max chain length |
|---|---|---|---|
| **Test 1** | 100 | 2.33 | 6 |
| **Test 2** | 100 | 2.44 | 6 |
| **Test 3** | 100 | 2.17 | 5 |
| **Test 4** | 100 | 2.22 | 4 |
| **Test 5** | 100 | 2.22 | 6 |

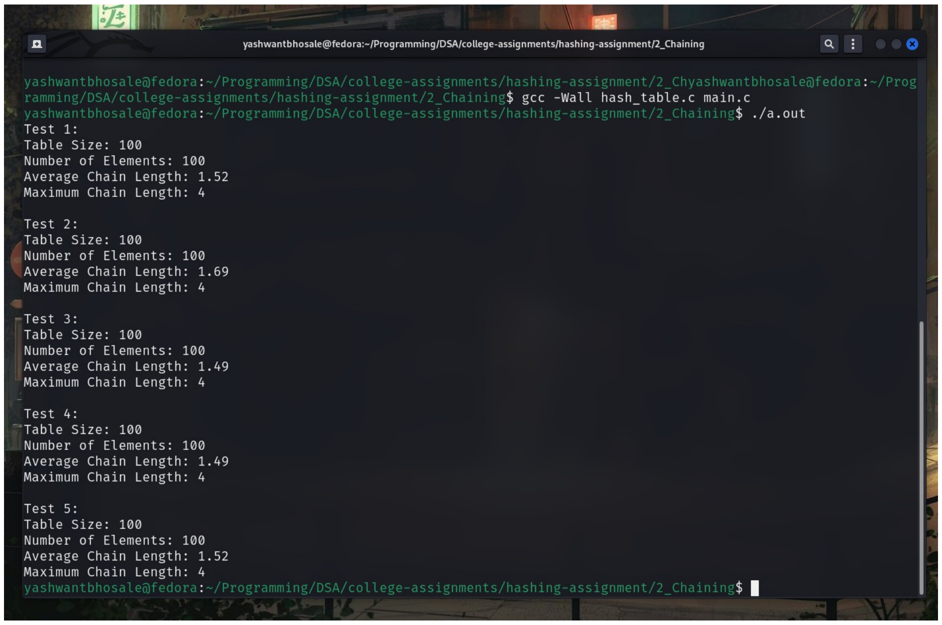


```
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/2_Chaining$ gcc -Wall ha
sh_table.c main.c
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/2_Chaining$ ./a.out
Test 1:
Table Size: 50
Number of Elements: 100
Average Chain Length: 2.33
Maximum Chain Length: 6

Test 2:
Table Size: 50
Number of Elements: 100
Average Chain Length: 2.44
Maximum Chain Length: 6

Test 3:
Table Size: 50
Number of Elements: 100
Average Chain Length: 2.17
Maximum Chain Length: 5

Test 4:
Table Size: 50
Number of Elements: 100
Average Chain Length: 2.22
Maximum Chain Length: 4

Test 5:
Table Size: 50
Number of Elements: 100
Average Chain Length: 2.22
Maximum Chain Length: 6

yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/2_Chaining$
```

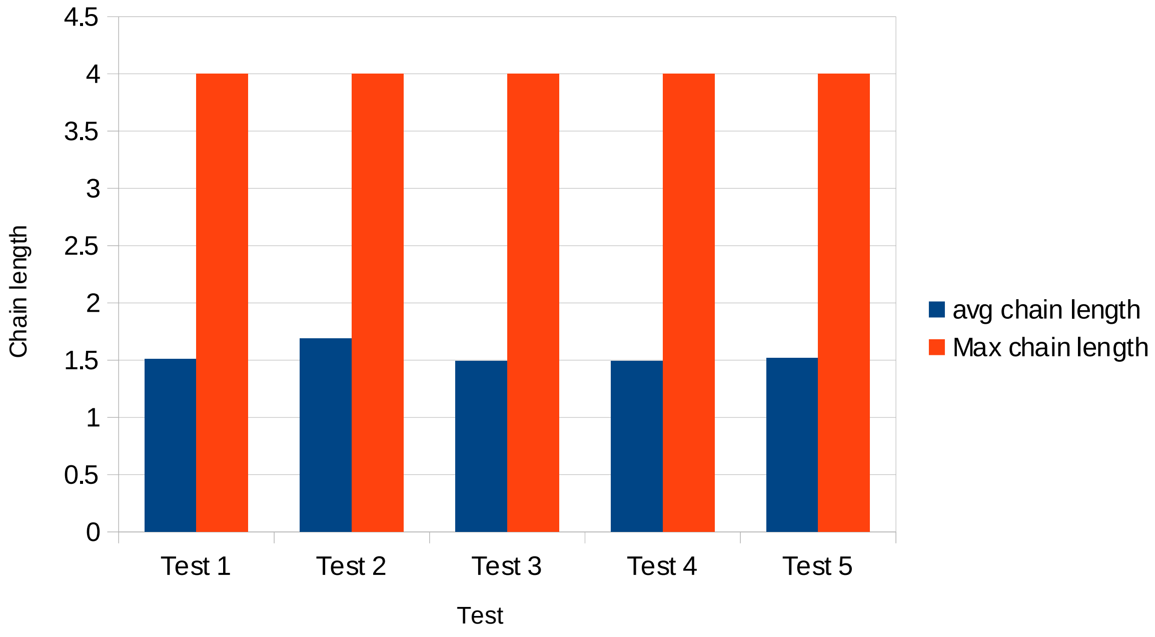


## 2) Table Size=100, Number of elements=100

| Tests | No. of elements | Avg chain length | Max chain length |
|---|---|---|---|
| **Test 1** | 100 | 1.52 | 4 |
| **Test 2** | 100 | 1.69 | 4 |
| **Test 3** | 100 | 1.49 | 4 |
| **Test 4** | 100 | 1.49 | 4 |
| **Test 5** | 100 | 1.52 | 4 |

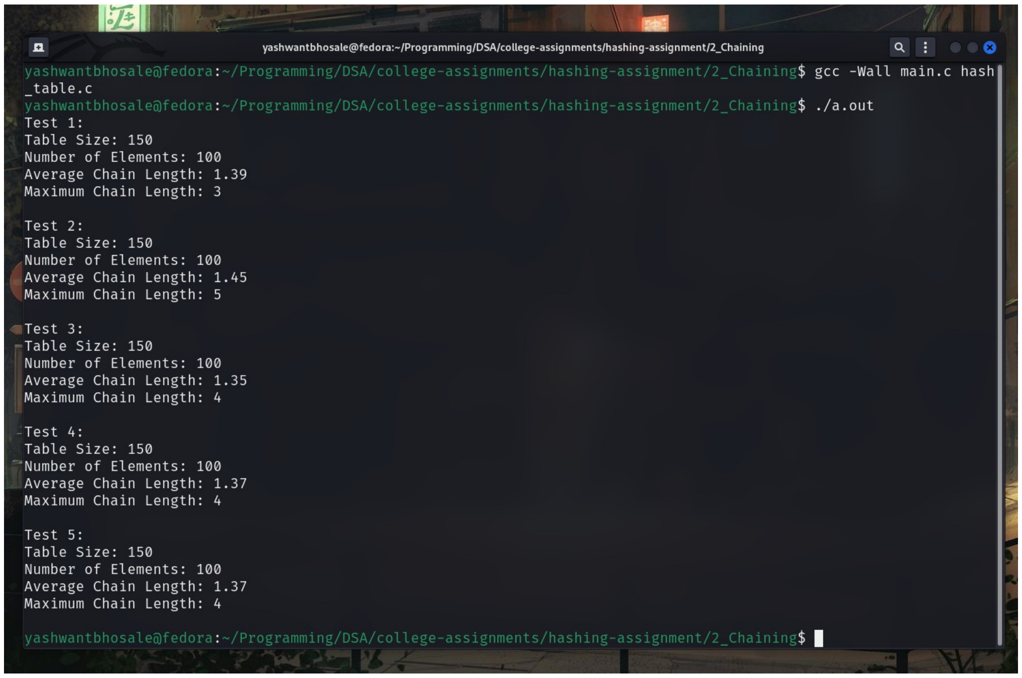


```
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/2_Chyashwantbhosale@fedora:~/Prog
ramming/DSA/college-assignments/hashing-assignment/2_Chaining$ gcc -Wall hash_table.c main.c
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/2_Chaining$ ./a.out
Test 1:
Table Size: 100
Number of Elements: 100
Average Chain Length: 1.52
Maximum Chain Length: 4

Test 2:
Table Size: 100
Number of Elements: 100
Average Chain Length: 1.69
Maximum Chain Length: 4

Test 3:
Table Size: 100
Number of Elements: 100
Average Chain Length: 1.49
Maximum Chain Length: 4

Test 4:
Table Size: 100
Number of Elements: 100
Average Chain Length: 1.49
Maximum Chain Length: 4

Test 5:
Table Size: 100
Number of Elements: 100
Average Chain Length: 1.52
Maximum Chain Length: 4
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/2_Chaining$
```

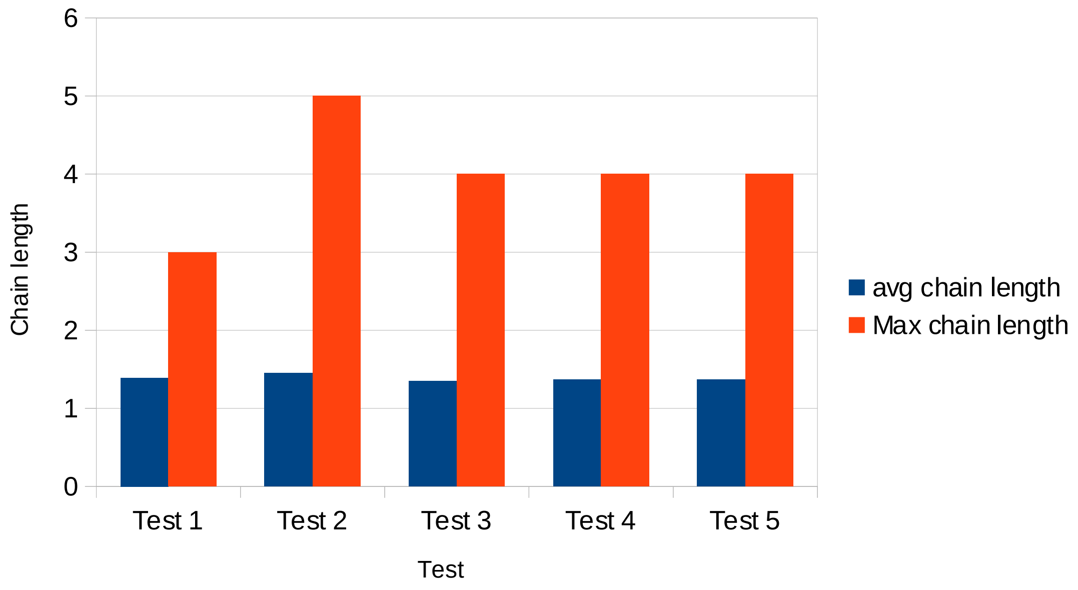


**3) Table size=150, Number of elements=100**

| Tests | No. of elements | Avg chain length | Max chain length |
|---|---|---|---|
| **Test 1** | 100 | 1.39 | 3 |
| **Test 2** | 100 | 1.45 | 5 |
| **Test 3** | 100 | 1.35 | 4 |
| **Test 4** | 100 | 1.37 | 4 |
| **Test 5** | 100 | 1.37 | 4 |



```
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/2_Chaining
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/2_Chaining$ gcc -Wall main.c hash_table.c
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/2_Chaining$ ./a.out
Test 1:
Table Size: 150
Number of Elements: 100
Average Chain Length: 1.39
Maximum Chain Length: 3

Test 2:
Table Size: 150
Number of Elements: 100
Average Chain Length: 1.45
Maximum Chain Length: 5

Test 3:
Table Size: 150
Number of Elements: 100
Average Chain Length: 1.35
Maximum Chain Length: 4

Test 4:
Table Size: 150
Number of Elements: 100
Average Chain Length: 1.37
Maximum Chain Length: 4

Test 5:
Table Size: 150
Number of Elements: 100
Average Chain Length: 1.37
Maximum Chain Length: 4

yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/2_Chaining$
```



**Driver code:**

**hash_table.c:**
```c
#include <stdio.h>
#include <stdlib.h>
#include "hash_table.h"

int hash(int key, int size) {
    return key % size;
}

void init_table(HashTable *ht, int size) {
    ht->size = size;
    ht->table = (Entry **)calloc(size, sizeof(Entry *));

    return;
}

void insert(HashTable *ht, int key, int value) {
    int index = hash(key, ht->size);

    Entry *new_entry = (Entry *)malloc(sizeof(Entry));
    new_entry->key = key;
    new_entry->value = value;
    new_entry->next = NULL;

    if(ht->table[index] == NULL) {
        ht->table[index] = new_entry;
    }else {
        Entry *cursor = ht->table[index];
        while(cursor->next != NULL) {
            cursor = cursor->next;
        }
        cursor->next = new_entry;
    }

    return;
}

void delete(HashTable *ht, int key) {
    int index = hash(key, ht->size);
    free(ht->table[index]);
    ht->table[index] = NULL;

    return;
}

int search(HashTable *ht, int key) {
    int index = hash(key, ht->size);
    int value = ht->table[index]->value;

    if (ht->table[index] == NULL) {
        return -1;
    }else {
        Entry *cursor = ht->table[index];
        while(cursor->next && cursor->key != key) {
            cursor = cursor->next;
        }
        value = cursor->value;
    }

    return value;
}

void print_table(HashTable *ht) {
    for (int i = 0; i < ht->size; i++) {
        if (ht->table[i] != NULL) {
            printf("{ Key: %d, Value: %d }\n", ht->table[i]-
>key, ht->table[i]->value);
        }else {
            printf("{ ___ }\n");
        }
    }
    return;
}
```

**main.c**

```
#include <stdio.h>
#include <stdlib.h>
#include "hash_table.h"

void calculate(HashTable *ht, double *avg_chain_length, int
*max_chain_length);
```

```c
int main() {
    int table_size = 150;
    int num_elements =100;

    for (int test = 1; test <= 5; test++) {
        HashTable ht;
        init_table(&ht, table_size);

        for (int i = 0; i < num_elements; i++) {
            int key = rand() % 1000;
            int value = rand() % 1000;
            insert(&ht, key, value);
        }

        double avg_chain_length;
        int max_chain_length;
        calculate(&ht, &avg_chain_length, &max_chain_length);

        // Print result
        printf("Test %d:\n", test);
        printf("Table Size: %d\n", table_size);
        printf("Number of Elements: %d\n", num_elements);
        printf("Average Chain Length: %.2f\n", avg_chain_length);
        printf("Maximum Chain Length: %d\n\n", max_chain_length);
    }

    return 0;
}

void calculate(HashTable *ht, double *avg_chain_length, int
*max_chain_length) {
    int total_chains = 0;
    int total_elements = 0;
    *max_chain_length = 0;

    for (int i = 0; i < ht->size; i++) {
        int chain_length = 0;
        Entry *current = ht->table[i];
        while (current) {
            chain_length++;
            current = current->next;
        }
        if (chain_length > *max_chain_length) {
            *max_chain_length = chain_length;
        }
        if (chain_length > 0) {
            total_chains++;
        }
        total_elements += chain_length;
    }

    *avg_chain_length = total_chains ? total_elements /
(double)total_chains : 0.0;
}
```

## 3. Double Hashing
Double hashing is a collision resolution technique used in hash tables. It works by using two hash functions to compute two different hash values for a given key. The first hash function is used to compute the initial hash value, and the second hash function is used to compute the step size for the probing sequence.

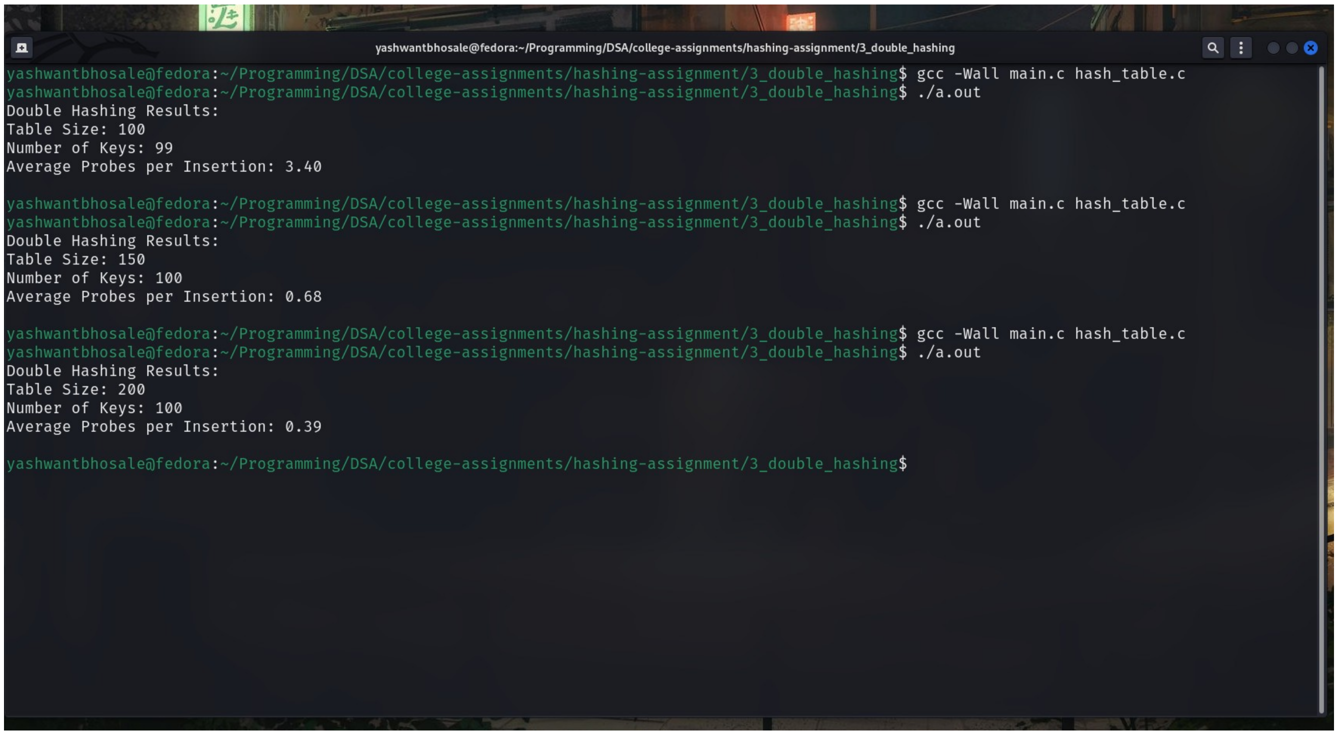
Double hashing has the ability to have a low collision rate, as it uses two hash functions to compute the hash value and the step size. This means that the probability of a collision occurring is lower than in other collision resolution techniques such as linear probing or quadratic probing.

However, double hashing has a few drawbacks. First, it requires the use of two hash functions, which can increase the computational complexity of the insertion and search operations. Second, it requires a good choice of hash functions to achieve good performance. If the hash functions are not well-designed, the collision rate may still be high.

Typical insert function using double hashing:
```
void insert(HashTable *ht, int key, int value) {
    unsigned h = hash(key, ht->size);
    unsigned h2 = hash_2(key, ht->size);
    unsigned i = 0;
    while (i < ht->size) {
        unsigned index = (h + i * h2) % ht->size; // use
secondary hash function for probing
        if (ht->table[index] == NULL) {
            Entry *entry = (Entry
*)malloc(sizeof(Entry));
            entry->key = key;
            entry->value = value;
            entry->hash = index;
            ht->table[index] = entry;
            ht->count++;
            return;
        }
        i++;
        ht->total_probes++;
    }
}
```

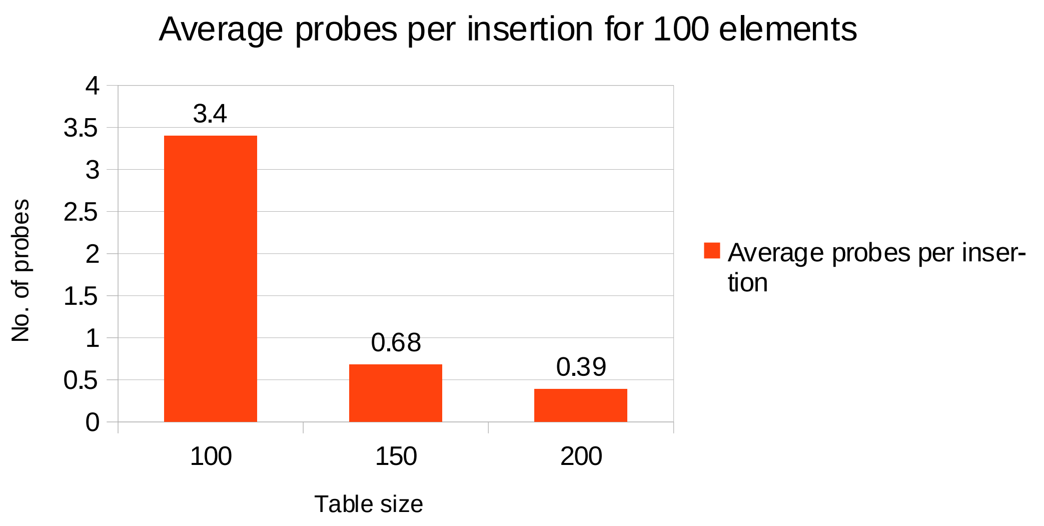
**Results:**

1. **Increase in Table Size**
   as the table size increases, the average probes per insertion drastically decrease. This is expected as fewer collisions occur with more slots available in the table reducing number of probes needed to find an empty slot.
2. **Efficiency Improvement**
   with larger table sizes double hashing performs much better in terms of probing which helps in minimizing the insertion time and improving overall efficiency.
3. **Comparison with chaining technique**
   Double Hashing may perform better as the table size increases. As the number of average probes per insertion reduce drastically with increase in table size.
   Also, we may fine tune the secondary hash function to complement the existing one so that both of them together perform at higher efficiency.
   In chaining however table size has little to no effect over the collisions as collisions are handled with linked list and we still have to traverse longer lists for search and insert operation.
   Chaining is simpler when it comes to complexity and implementation but performance and efficiency wise **double hashing is more efficient.**



```
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/3_double_hashing$ gcc -Wall main.c hash_table.c
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/3_double_hashing$ ./a.out
Double Hashing Results:
Table Size: 100
Number of Keys: 99
Average Probes per Insertion: 3.40

yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/3_double_hashing$ gcc -Wall main.c hash_table.c
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/3_double_hashing$ ./a.out
Double Hashing Results:
Table Size: 150
Number of Keys: 100
Average Probes per Insertion: 0.68

yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/3_double_hashing$ gcc -Wall main.c hash_table.c
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/3_double_hashing$ ./a.out
Double Hashing Results:
Table Size: 200
Number of Keys: 100
Average Probes per Insertion: 0.39

yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/3_double_hashing$
```

| **Test** | **Table size** | **Number of elements** | **Avg. probes per insertion** |
|---|---|---|---|
| **Test 1** | 100 | 100 | 3.40 |
| **Test 2** | 150 | 100 | 0.68 |
| **Test 3** | 200 | 100 | 0.39 |



**Driver code:**

**hash_table.c**

```
#include <stdio.h>
#include <stdlib.h>
#include "hash_table.h"


void init_table(HashTable *ht, int size) {
    ht->table = (Entry **)malloc(size * sizeof(Entry *));
    ht->count = 0;
    ht->size = size;
    ht->total_probes = 0;
    for (int i = 0; i < size; i++) {
        ht->table[i] = NULL;
    }
}

unsigned hash(int key, int size) {
    return key % size;
}

unsigned hash_2(int key, int size) {
    return 1 + (key % (size - 1));
}

void insert(HashTable *ht, int key, int value) {
    unsigned h = hash(key, ht->size);
    unsigned h2 = hash_2(key, ht->size);
    unsigned i = 0;

    while (i < ht->size) {
        unsigned index = (h + i * h2) % ht->size; // use secondary
hash function for probing
        if (ht->table[index] == NULL) {
            Entry *entry = (Entry *)malloc(sizeof(Entry));
            entry->key = key;
            entry->value = value;
            entry->hash = index;
            ht->table[index] = entry;
            ht->count++;

            return;
        }
        ht->total_probes++;
        i++;
    }
}
```

**main.c:**

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "hash_table.h" // Include your hash table header file

#define NUM_KEYS 100

void generate_random_data(int keys[], int values[], int num_keys) {
    for (int i = 0; i < num_keys; i++) {
        keys[i] = rand() % 1000;
        values[i] = rand() % 100;
    }
}

int main() {
    srand(time(NULL)); // Seed random number generator

    int keys[NUM_KEYS];
    int values[NUM_KEYS];
    generate_random_data(keys, values, NUM_KEYS);

    HashTable ht;
    init_table(&ht, 200);

    for (int i = 0; i < NUM_KEYS; i++) {
        insert(&ht, keys[i], values[i]);
    }

    printf("Double Hashing Results:\n");
    printf("Table Size: %d\n", ht.size);
    printf("Number of Keys: %d\n", ht.count);
    printf("Average Probes per Insertion: %.2f\n",
(double)ht.total_probes / (double)ht.count);
    printf("\n");

    return 0;
}
```

## 4. Linear and Quadratic Probing (Open addressing)

**Open Addressing** is a method for handling collisions. In Open Addressing, all elements are stored in the **hash table** itself. So at any point, the size of the table must be greater than or equal to the total number of keys (we may increase table size by copying old data if needed).

### Linear Probing:

In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

So, basically once a collision occurs we have to traverse the table linearly just like the linear search.

### Quadratic Probing

It is interesting to note that the intervals between probes increase proportionally to the hash value. Quadratic probing is a method with which we take advantage of this behaviour and also avoid the clustering of data around the index where lot of collisions occur. This method is also known as the mid-square method. In this method we look for $i^2$th slot for ith iteration. We always start from the original hash location. If the location is occupied then only we check for other slots.

Linear Probing Insert Function:

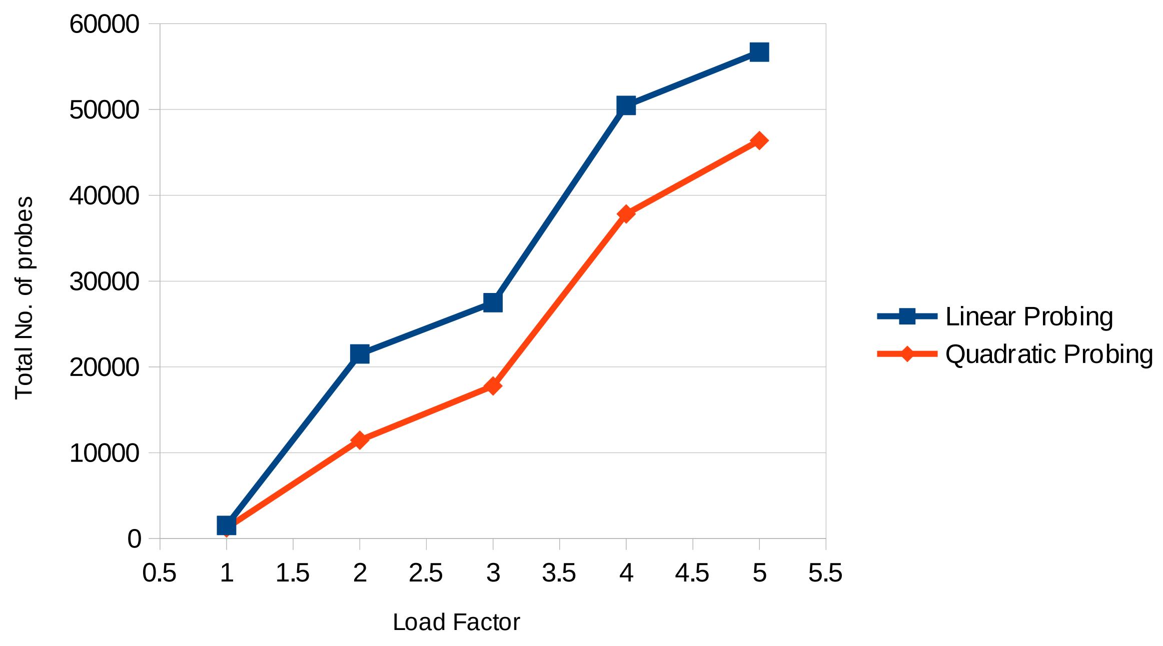
```c
void insert(HashTable *ht, int key, int value) {
    unsigned hash_t = hash(key, ht->size);
    int index = hash_t;
    unsigned probes = 0;

    while (ht->table[index] != NULL) {
        index = (index + 1) % ht->size;
        probes++;
    }

    ht->table[index] = (Entry *)malloc(sizeof(Entry));
    ht->table[index]->key = key;
    ht->table[index]->value = value;
    ht->count++;

    ht->total_probes += probes;

    return;
}
```

Quadratic Probing insert function:

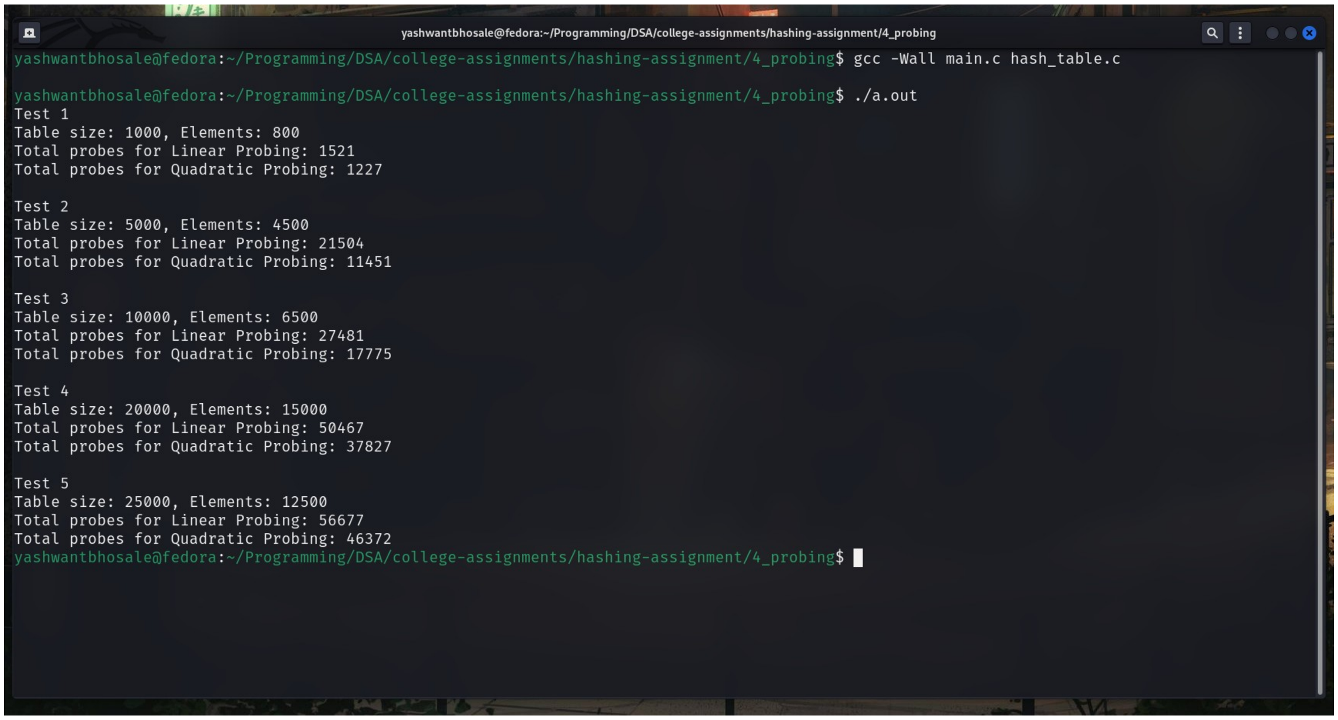
```c
void insert_quadratic_probing(HashTable *ht, int key, int
value) {
    if (ht->count == ht->size) {
        return;
    }

    unsigned hash_t = hash(key, ht->size);
    int index = hash_t;
    unsigned probes = 0;

    for (int i = 0; ht->table[index] != NULL && ht-
>table[index]->key != key; i++) {
        index = (hash_t + i * i) % ht->size;
        probes++;

        if (probes >= ht->size) {
            return;
        }
    }

    Entry *new_entry = (Entry *)malloc(sizeof(Entry));
    if (new_entry == NULL) {
        return;
    }

    new_entry->key = key;
    new_entry->value = value;
    ht->table[index] = new_entry;
    ht->count++;

    ht->total_probes += probes;

}
```

**Results:**

1. **Perfomance and load factor**
   as the number of elements increases,compared to the table size, both probing methods show an increase in the total probes required
   Linear Probing tends to have a higher to tal number of quadratic probing in this trend. Also when the load factor increases linear probing shows drastic increase in number of total probes compared to quadratic probing.
2. **Efficiency**
   as per the results quadratic probing consistently performed better than the linear one as it requires fewer probes than linear probing across all tests. This shows that quadratic probic has better efficiency with larger load factor and table sizes.
3. **Impact of table sizes**
   higher table sizes genrally reduce the total number of probes required for both the methods.
   However, Linear Probing performs less efficiently as load factor increases, as a result we get more collisions and longer chains rather a cluster of elements with same hash value.
   In quadratic probing increase in probes is lesser which indicates its efficiency.
4. **Trend**
   For both the methods number of collisions increase in non linear fashion.



| Test | Table size | Elements | Load factor | Linear Probing probes | Quadratic probing Probes |
|---|---|---|---|---|---|
| **1** | 1000 | 800 | 0.8 | 1521 | 1227 |
| **2** | 5000 | 4500 | 0.90 | 21504 | 11451 |
| **3** | 10000 | 6500 | 0.65 | 27481 | 17775 |
| **4** | 15000 | 20000 | 0.75 | 50467 | 37827 |
| **5** | 25000 | 12500 | 0.50 | 56677 | 46372 |



yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/4_probing

```
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/4_probing$ gcc -Wall main.c hash_table.c

yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/4_probing$ ./a.out
Test 1
Table size: 1000, Elements: 800
Total probes for Linear Probing: 1521
Total probes for Quadratic Probing: 1227

Test 2
Table size: 5000, Elements: 4500
Total probes for Linear Probing: 21504
Total probes for Quadratic Probing: 11451

Test 3
Table size: 10000, Elements: 6500
Total probes for Linear Probing: 27481
Total probes for Quadratic Probing: 17775

Test 4
Table size: 20000, Elements: 15000
Total probes for Linear Probing: 50467
Total probes for Quadratic Probing: 37827

Test 5
Table size: 25000, Elements: 12500
Total probes for Linear Probing: 56677
Total probes for Quadratic Probing: 46372
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/hashing-assignment/4_probing$
```

Driver Code:
**hash_table.c**
```c
// hash table: linear probing
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "hash_table.h"

void init_table(HashTable *ht, int size) {
    ht->table = (Entry **)calloc(size, sizeof(Entry *));
    ht->size = size;
    ht->count = 0;
    return;
}

/*
unsigned hash (int key) {
    unsigned hash = 0;
    srand(time(NULL));

    for (int i = 0; i < 4; i++) {
        hash = (hash << 8) | (rand() & 0xFF);
    }

    return hash;
}
*/

unsigned hash(int key, int size) {
    return key % size;
}

void insert(HashTable *ht, int key, int value) {
    unsigned hash_t = hash(key, ht->size);
    int index = hash_t;
    unsigned probes = 0;

    while (ht->table[index] != NULL) {
        index = (index + 1) % ht->size;
        probes++;
    }

    ht->table[index] = (Entry *)malloc(sizeof(Entry));
    ht->table[index]->key = key;
    ht->table[index]->value = value;
    ht->count++;

    ht->total_probes += probes;

    return;
}

void delete(HashTable *ht, int key) {
    unsigned index = hash(key, ht->size);

    while (ht->table[index] != NULL) {
        if (ht->table[index]->key == key) {
            free(ht->table[index]);
            ht->table[index] = NULL;
            ht->count--;
            return;
        }
        index = (index + 1) % ht->size;
    }

    return;
}

int search(HashTable *ht, int key) {
    unsigned index = hash(key, ht->size);

    while (ht->table[index] != NULL) {
        if (ht->table[index]->key == key) {
            return ht->table[index]->value;
        }
        index = (index + 1) % ht->size;
    }
    return -1;
```

```c
}

void print_table(HashTable *ht) {
    for (int i = 0; i < ht->size; i++) {
        if (ht->table[i] != NULL) {
            printf("Index: %d, Key: %d, Value: %d\n", i,
ht->table[i]->key, ht->table[i]->value);
        }
    }
    return;
}

void insert_quadratic_probing(HashTable *ht, int key, int
value) {
    if (ht->count == ht->size) {
        return;
    }

    unsigned hash_t = hash(key, ht->size);
    int index = hash_t;
    unsigned probes = 0;

    for (int i = 0; ht->table[index] != NULL && ht-
>table[index]->key != key; i++) {
        index = (hash_t + i * i) % ht->size;
        probes++;

        if (probes >= ht->size) {
            return;
        }
    }

    Entry *new_entry = (Entry *)malloc(sizeof(Entry));
    if (new_entry == NULL) {
        return;
    }

    new_entry->key = key;
    new_entry->value = value;
    ht->table[index] = new_entry;
    ht->count++;

    ht->total_probes += probes;

}
```

**main.c**

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "hash_table.h"

#define NUM_TESTS 5

int main() {
    HashTable ht1, ht2;

    srand(time(NULL));
    int table_sizes[NUM_TESTS] = {1000, 5000, 10000,
20000, 25000};
    int num_elements[NUM_TESTS] = {800, 4500, 6500,
15000, 12500};

    for(int i = 0; i < NUM_TESTS; i++) {
        init_table(&ht1, table_sizes[i]);
        init_table(&ht2, table_sizes[i]);

        for (int j = 0; j < num_elements[i]; j++) {
            insert(&ht1, rand(), j);
        }

        for (int j = 0; j < num_elements[i]; j++) {
            insert_quadratic_probing(&ht2, rand(), j);
        }

        printf("Test %d\n", i + 1);
        printf("Table size: %d, Elements: %d\n",
table_sizes[i], num_elements[i]);
```

```
        printf("Total probes for Linear Probing: %u\n",
ht1.total_probes);
        printf("Total probes for Quadratic Probing: %u\
n", ht2.total_probes);
        printf("\n");
    }
    return 0;
}
```

**Final Conclusions:**

1. Hashing is a powerful technique used for efficient data retrieval, with different methods for handling collisions. The performance and efficiency of a hash table depend on the chosen method, which can significantly impact both insertion and searching operations.
2. **Simple hash function:** like modulo is easy to implement it does not handle the classic case of collision where two keys get mapped to same index so we have to employ some of the many available techniques for handling collision.
3. **Chaining:** In this technique, we deal with collisions by chaining the colliding records using linked list. While it is simple and easy to implement it is difficult to achieve efficiency.
4. **Double Hashing:** It is sort of a generalized form of open addressing it uses two functions to reduce clustering and is more efficient than linear probing and chaining for dealing with collisions. We can fine tune to functions to compliment each other and have a more uniform distribution of records and thus a better performance.
5. **Open addressing:** In open addressing, all elements are stored within the table but we probe to find next location when a collision occurs. Among two discussed techniques, that is, linear and quadratic probing quadratic probing consistently performs better with larger load factors and table sizes.
6. **Searching:** Searching in a hash table is closely related to insertion and collision resolution technique. This report doesn't deeply discuss searching as it is done in very similar way as insertion. For methods like chaining we traverse the chain, and for methods like open addressing we probe similar to the way we probe during insertion.

In conclusion, open addressing methods like double hashing provide a good balance between space efficiency and speed, while chaining offers simplicity at the cost of space. The choice of technique depends on the specific use case and the expected load on the hash table