CS 584/684 Algorithm Design and Analysis

Final Project

Hashing and Collision Resolution Techniques

Srivashinie Dhamodharasamy (954814608)

Table of Contents

What is hashing?

Hashing is a technique for mapping data, typically for efficient storage and retrieval in data structures such as hash tables. It involves using a hash function to convert input data into a hash code, which is then used to index or locate data within the data structure. Hashing is used to optimize data retrieval operations because it allows for quick access to data by narrowing the search space to a single location based on the hash code.

Why Hashing?

In today's data-intensive internet and programming scenario, efficient storage, access, and processing are important. Conventional data structures such as arrays, which require at least O (log n) time for search operations, can be inefficient for large datasets, even though they offer O(1) time complexity for storage. The inability to store and retrieve data in constant time led to the design of the hashing data structure. Using a hash function, hashing provides an efficient way to implement key-value data structures, ensuring quick access by mapping keys to table indices. This leads to constant-time complexity (O (1)) for retrieval operation. By assigning a fixed space for storing elements, hashing reduces memory usage, which is especially advantageous for programs with limited memory. Hashing works well with large datasets, ensuring consistent access time as the dataset grows. Overall, hashing is significant for managing and storing data effectively.

Components of Hashing

- **Hash Table :** A hash table is a data structure that stores key-value pairs by mapping keys to their corresponding values using a hash function. It is a generalized array in which each element is stored in a position determined by its key.
- **Key :** A key is passed as input to the hash function. The hash function then computes the index or location in a data structure where the corresponding item will be stored.
- **Hash Function :** The keys in the array are mapped to integer indices using a hash function. Each key should map to a unique index while minimizing collisions or mapping multiple keys to the same index. It is also important to ensure that the hash function is simple to compute and evenly distributes elements in the table. Some of the commonly used hash functions are:
  - **Division Method :** The Division Method divides the key by the table size and assigns the remainder as the index.

    $$h(k)=k \bmod m$$
  - **Mid Square Method :** The Mid Square Method squares the key and uses the middle digits as the index, which provides good performance but is limited by the key size.

    $$h(k)=h(k \times k)$$
  - **Folding Method :** The Digit Folding Method divides the key into parts and sums them, allowing flexibility in key size.

    $$h(k) = k_1 + k_2 + \cdots + k_n$$

    where k1,k2…kn are part has the same number of digits except for the last part which can have lesser digits than the other parts
  - **Multiplication Method :** The Multiplication Method involves multiplying the key by a constant, extracting the fractional part, and multiplying by the table size M

    $$h(K)=\text{floor}(M \times (k \times A \bmod 1))$$
- **Collisions and Collision Resolution Techniques:** Hash functions are necessary for mapping data based on unique keys. However, collisions can happen when two different keys are mapped to the same hash value due to a small range of hash values compared to the infinite number of keys. Collision resolution

techniques are then used to solve the problem by mapping alternative locations for colliding keys. There are generally two types of collision resolution techniques:

- Open hashing.
- Closed hashing.

Pseudo code of Hashing:

- **Initialize:** First create a hash table of size M .
- **Insert(key):**
  - Compute the hash value of the key using a hash function, which returns an index in the range [0, M-1].
  - Add the key to the table at the computed index.
- **Find(key):**
  - Compute the hash value of the key using the same hash function.
  - Iterate through the list at the computed index to find the key. If the key is found, return it. Else, return not found.
- **Delete(key):**
  - Compute the hash value of the key using the same hash function.
  - Iterate through the list at the computed index to find the key. If the key is found, remove it from the list. Else, return not found.

```
Initialize(M):
   table = new Array of size M
   return table

Insert(table, K, hash_func):
   index = hash_func(K)
   table[index]= K  //add the key to hash table at the computed index


Find(table, K, hash_func):
   index = hash_func(K)
   for key in table[index]:
      if key == K:
         return key
   return -1   //indicating the key is not found


Delete(table, K, hash_func):
   index = hash_func(K)
   if table[index] == K:
         table[index].remove(K)
         return
   return -1  //indicating the key is not found
```

Open hashing

In open hashing, each hash table slot contains a linked list of elements that hash to the same slot. When two keys hash to the same position in the hash table, both keys are stored in that location. However, the new key is appended to a linked list that is linked to that position rather than replacing the current one. This linked list enables multiple keys to be stored in the same position in the hash table, effectively preventing collisions. This method gives the technique its name because it uses linked lists to create a chain of keys at each collision index.

Pseudo Code:

```
chaining_insert(K):
  index = h(K)
  if table[index] is None:
    table[index] = [K]
  else:
    table[index].append(K)
```

Time complexity: Searching, insertion, and deletion have O (1 + k/M) average complexity, where k is the number of elements and M is the size of the table.

Closed Hashing

Without the need for extra data structures, closed hashing aims to store every element within the hash table itself. When a collision occurs in closed hashing, the algorithm searches the hash table for the next available empty slot. There are several methods for searching, like linear probing, quadratic probing, and double hashing.

- **Linear Probing:** When a collision happens and a slot is already taken, the linear probing algorithm sequentially looks for the next open slot. It checks the next slot and keeps checking until it finds an empty slot.

  The formula for linear probing is rehash(k) = (hash(k)+1) % table size

  Pseudo code:

  ```
  linear_probing_insert(K):
    if table is full:
      return error
    probe = h(K)
    while table[probe] is occupied:
      probe = (probe + 1) mod M
    table[probe] = K
  ```

  Time complexity: Searching, insertion, and deletion have O (1) average complexity, but can degrade to O(n) in worst cases.
- **Quadratic Probing:** Using a quadratic function, the algorithm determines the next probe position in quadratic probing as opposed to probing the next slot in a linear manner. A quadratic pattern is followed by the probing sequence until an empty slot is located. Quadratic probing is also referred to as the mid-square method.

  The formula for Quadratic Probing is rehash(k) = (hash(k) + c1.i +c2.i^2) % table size

  Pseudo code:

  ```
  quadratic_probe_insert(K):
    if table is full:
  ```

```
    return error
  probe = h(K)
  i = 1
  while table[probe] is occupied:
    probe = (probe + i^2) mod M
    i = i + 1
  table[probe] = K
```

Time complexity: Searching, insertion, and deletion have O(1) average complexity, but can degrade to O(n) in worst cases.

- **Double hashing:** Double hashing finds the probe sequence by using two hash functions. The next slot to probe is determined by applying the second hash function to compute an offset in the case of a collision. The formula for Double hashing is rehash(k) = (hash(k) + hash2(k)) % table size

  Pseudo code:

```
double_hash_insert(K):
  if table is full:
    return error
  probe = h1(K)
  offset = h2(K)
  while table[probe] is occupied:
    probe = (probe + offset) mod M
  table[probe] = K
```

  Time complexity: Searching, insertion, and deletion have O(1) average complexity, but performance can degrade if hash functions produce many collisions.

Load Factor

The load factor of a hash table is a measure of how full the hash table is. It is calculated as the number of elements stored in the hash table divided by the size of the hash table. A high load factor means that the hash table is more likely to have collisions, where two keys hash to the same index. A low load factor means that the hash table has plenty of empty slots, which can be wasteful in terms of memory. In general, a load factor between 0.5 and 0.7 is considered optimal for a hash table. This provides a good balance between space usage and performance.

Rehashing:

Rehashing is a process in hash tables that prevents collisions while maintaining efficiency. The load factor of a hash table increases as new elements are added. Performance is decreased because collisions are more likely to occur when the load factor rises above a particular threshold. All elements are rehashed to new buckets based on the resized hash table. As a result, performance is improved, and the load factor and collision frequency are decreased. Rehashing is required to keep the hash table as efficient as possible, but it can be time-consuming and memory intensive.

Experiment and Results

Random input values were generated for every experiment conducted, and a hash table of the specified size was created. The chosen collision resolution technique was then used to insert the input values into the hash table, and the execution times of each operation were recorded. The execution times are measured in seconds. To avoid problems with garbage collection, efforts were made to reduce unnecessary memory allocations and deallocations. Before beginning each experiment, the input values and hash table were also correctly initialized. Google Collab was used for the experiments to prevent interference from background processes.

Experiment 1 : Comparing the performance of different collision resolution techniques in hash tables for varying table sizes.

For each table size, a list of random input values was generated. This list represented a random selection of the values. The hash function used in the code is a simple division operation. For each table size, the program creates a new Hash Table object of that size. For each collision resolution technique (linear probing, quadratic probing, double hashing, and chaining), the program measures the execution time to insert the generated input values into the hash table using that technique. The program prints the average execution times for each technique and table size, and it plots a graph showing how the execution time varies with the table size for each technique.

Results:

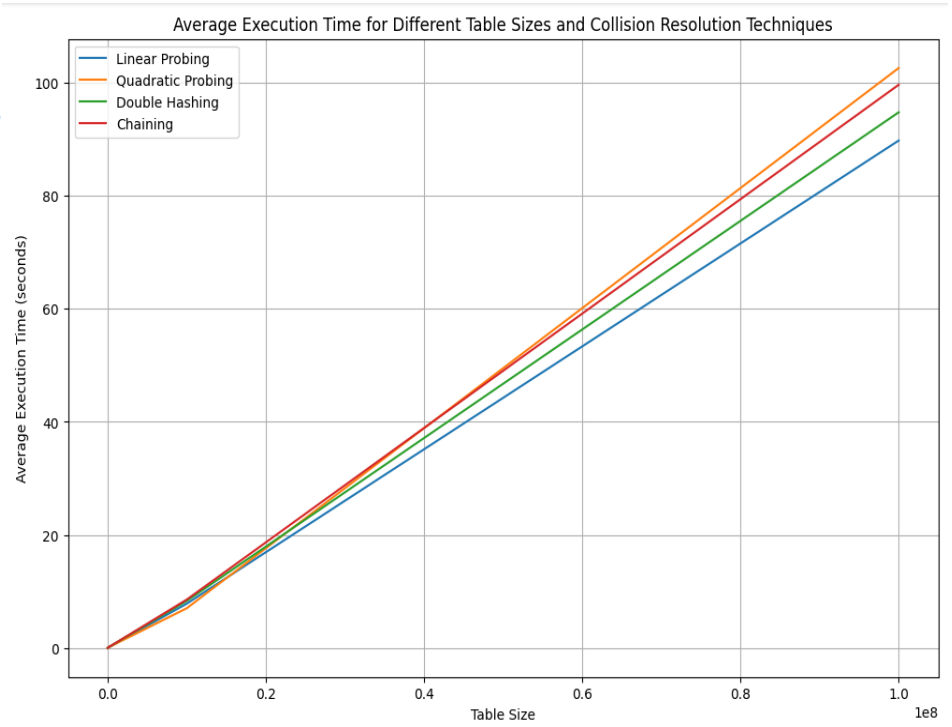
```
Table Size: 100
Linear Probing: 6.985664367675781e-05 seconds
Quadratic Probing: 6.222724914550781e-05 seconds
Double Hashing: 6.771087646484375e-05 seconds
Chaining: 7.510185241699219e-05 seconds

Table Size: 1000
Linear Probing: 0.0006830692291259766 seconds
Quadratic Probing: 0.001984119415283203 seconds
Double Hashing: 0.0008733272552490234 seconds
Chaining: 0.0009148120880126953 seconds

Table Size: 100000
Linear Probing: 0.09572267532348633 seconds
Quadratic Probing: 0.09754276275634766 seconds
Double Hashing: 0.11004209518432617 seconds
Chaining: 0.11706757545471191 seconds

Table Size: 10000000
Linear Probing: 7.842092037200928 seconds
Quadratic Probing: 7.030624151229858 seconds
Double Hashing: 8.329639196395874 seconds
Chaining: 8.562960863113403 seconds

Table Size: 100000000
Linear Probing: 89.7676591873169 seconds
Quadratic Probing: 102.59825682640076 seconds
Double Hashing: 94.76482510566711 seconds
Chaining: 99.63155579566956 seconds
```



Observations:

The results show the execution times for inserting elements into hash tables of different sizes using various collision resolution techniques: linear probing, quadratic probing, double hashing, and chaining.

For smaller table sizes, linear probing tends to have the lowest average execution times compared to quadratic probing, double hashing, and chaining. We can see a clear trend of increasing execution times as the table size grows exponentially while using linear probing. This indicates that linear probing becomes less efficient with larger table sizes, likely due to an increase in collisions and the need to perform more iterations to find an

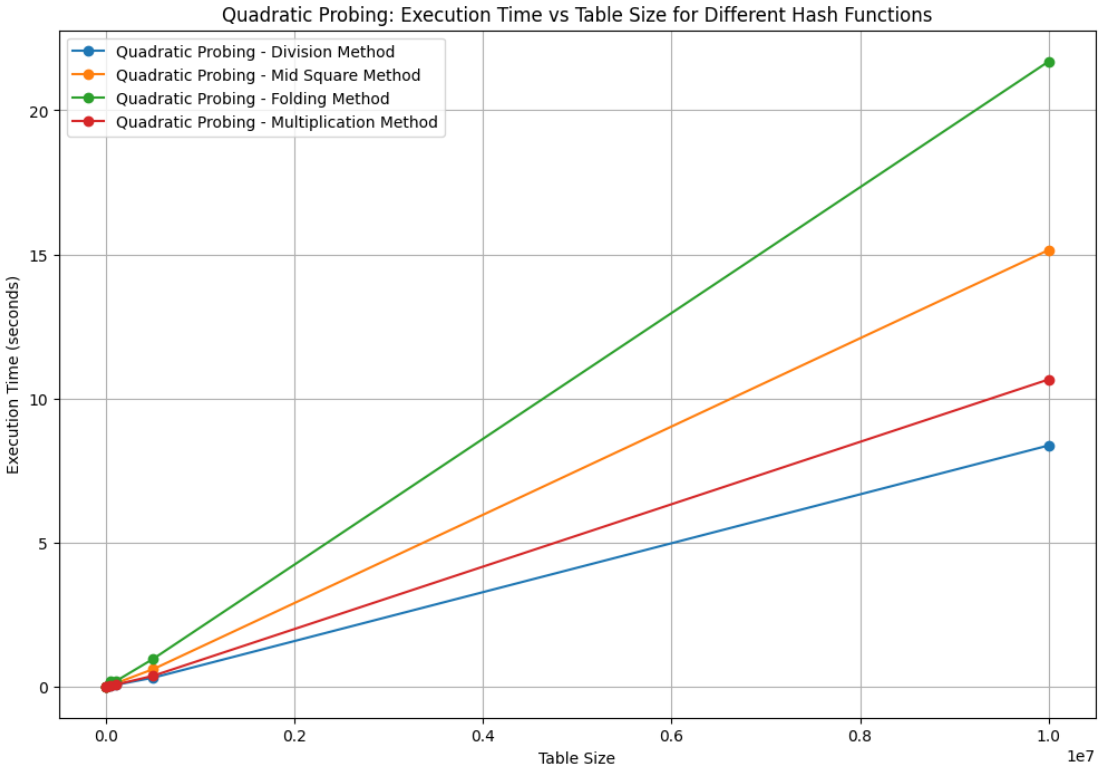
empty slot. In contrast, when we look at the execution times for quadratic probing and double hashing at the same table sizes, we see a more stable or slightly increasing trend, indicating that these methods are better able to handle the increased load and maintain more consistent performance as the table size increases.

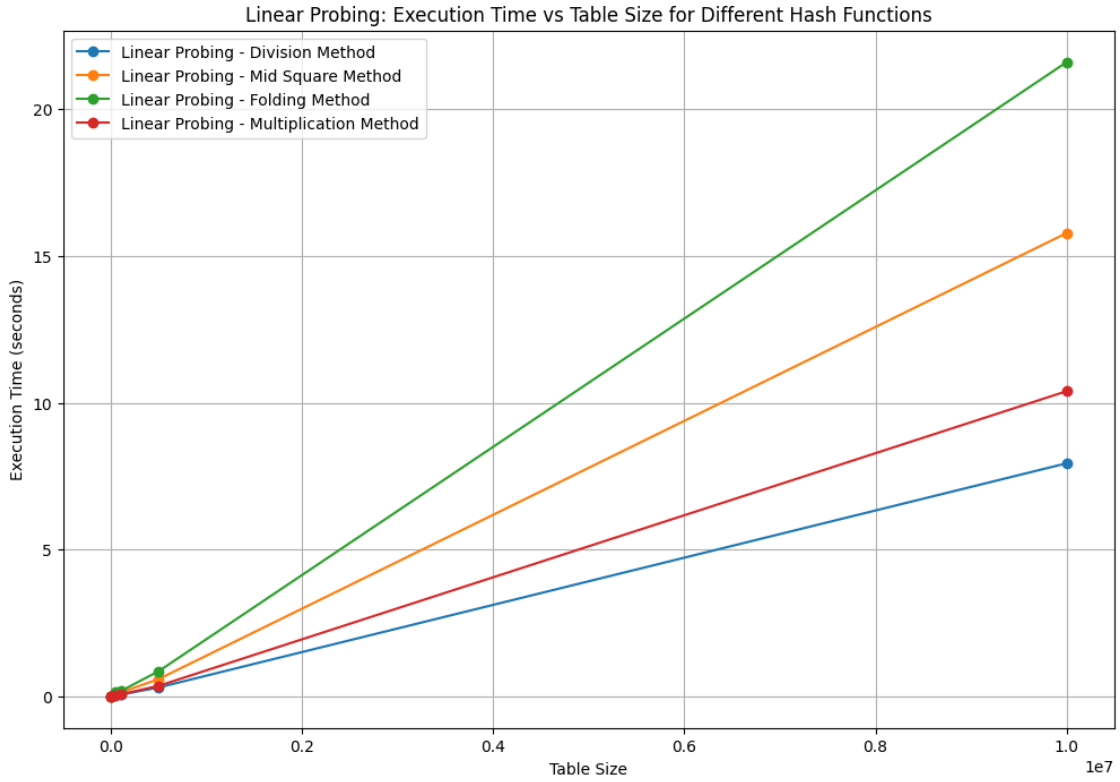
Experiment 2: Comparing the performance of different hash functions and collision resolution techniques in hash tables for varying table sizes.

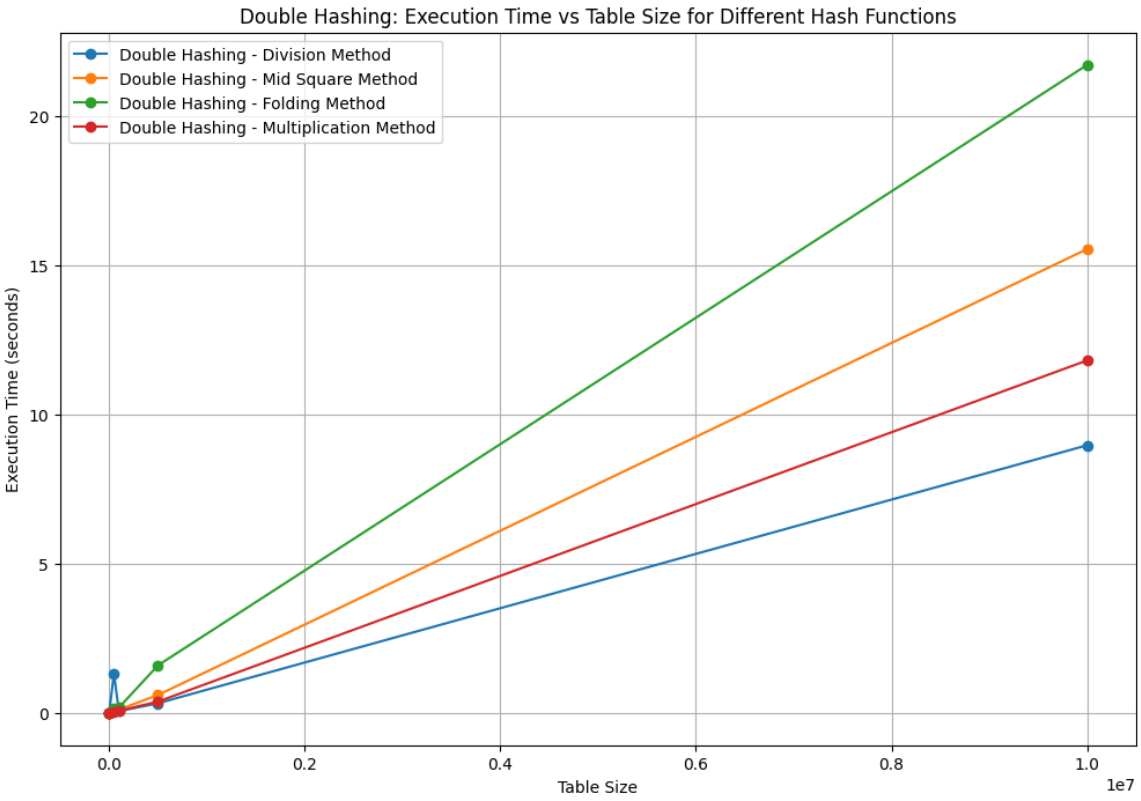
Four hash functions are implemented: Division Method, Mid Square Method, Folding Method, and Multiplication Method. Four collision resolution techniques are implemented: Linear Probing, Quadratic Probing, Double Hashing and Chaining. For each table size (100, 1000,50000, 100000,500000 ,10000000), the program generates input values and inserts them into a hash table using each combination of hash function and collision resolution technique. It measures the execution time for each combination and stores the results.

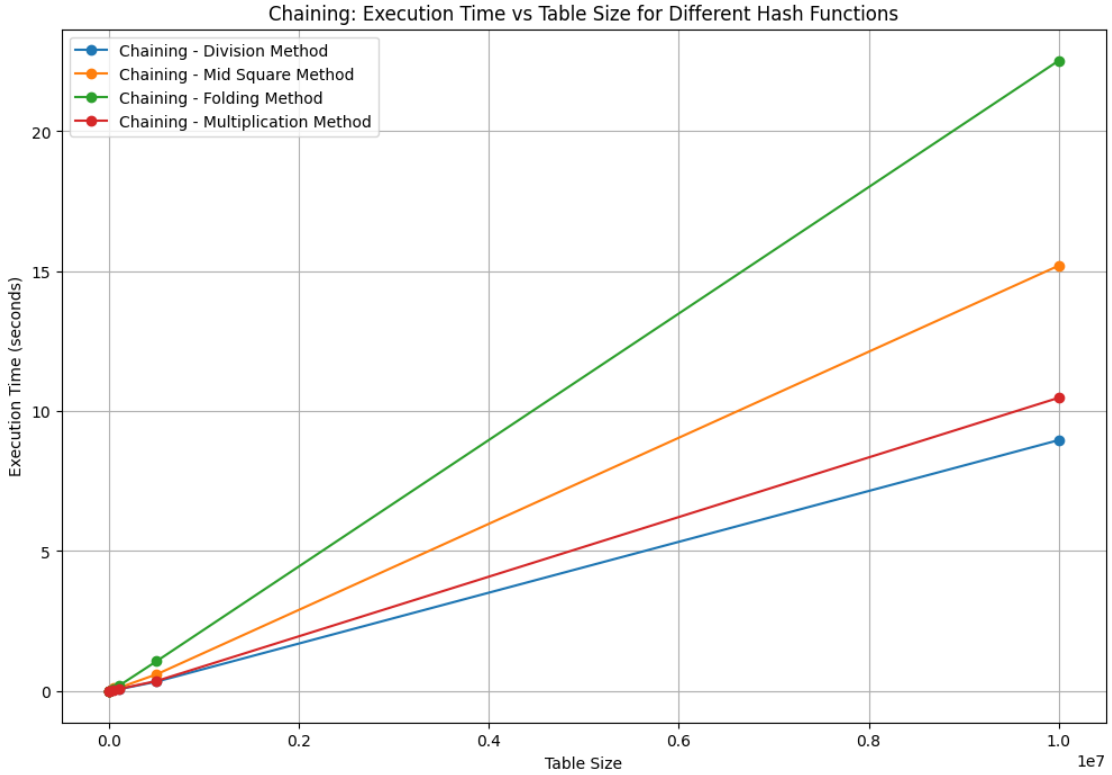
Results:

Hash Function	Probing Method	Execution Time
Division Method	Linear Probing	[8.273124694824219e-05, 0.000751495361328125, 0.047302961349487305, 0.04993438720703125, 0.2987821102142334, 7.936540603637695]
Mid Square Method	Linear Probing	[0.000143764031982422, 0.001890420913696289, 0.10394287109375, 0.12350797653198242, 0.5857250690460205, 15.783108949661255]
Folding Method	Linear Probing	[0.0002493858337402344, 0.003162384033203125, 0.1598987579345703, 0.17165613174438477, 0.8549292087554932, 21.607553958892822]
Multiplication Method	Linear Probing	[8.749961853027344e-05, 0.0009295940399169922, 0.029005765914916992, 0.06189680099487305, 0.3543210029602051, 10.394061326980591]
Division Method	Quadratic Probing	[8.463859558105469e-05, 0.0008170604705810547, 0.04576921463012695, 0.05204272270202637, 0.30393242835998535, 8.364701747894287]
Mid Square Method	Quadratic Probing	[0.00014019012451171875, 0.001897573471069336, 0.10681629180908203, 0.10712790489196777, 0.6008899211883545, 15.148450851440043]
Folding Method	Quadratic Probing	[0.0002841949462890625, 0.002948522567749023 4, 0.1794900894165039, 0.17690300941467285, 0.9594192504882812, 21.69303607940674]
Multiplication Method	Quadratic Probing	[8.916854858398438e-05, 0.0008862018585205078, 0.027996063232421875, 0.06179356575012207, 0.37058234214782715, 10.655133485794067]
Division Method	Double Hashing	[9.059906005859375e-05, 0.0009367465972900391, 1.314101219177246, 0.05321907997131348, 0.31830906867980957, 8.968482255935669]
Mid Square Method	Double Hashing	[0.0001342296600341797, 0.001891374588012695 3, 0.09993600845336914, 0.10497331619262695, 0.5946304798126221, 15.546665191650391]
Folding Method	Double Hashing	[0.00025916099548339844, 0.0030667781829833984, 0.1564192771911621, 0.17145729064941406, 1.589669942855835, 21.72097611427307]
Multiplication Method	Double Hashing	[8.511543273925781e-05, 0.0008928775787353516, 0.0277559757232666, 0.0652930736541748, 0.37450575828552246, 11.813023567199707]
Division Method	Chaining	[9.369850158691406e-05, 0.0009481906890869141, 0.05989241600036621, 0.05685925483703613, 0.3323829174041748, 8.961991548538208]
Mid Square Method	Chaining	[0.0001444816589355468 8, 0.0019295215606689453, 0.1093454360961914, 0.1101841926574707, 0.5939948558807373, 15.196150779724121]
Folding Method	Chaining	[0.0002603530883789062 5, 0.003103494644165039, 0.084503173828125, 0.17943882942199707, 1.065180778503418, 22.522632360458374]
Multiplication Method	Chaining	[0.0001037120819091796 9, 0.0008890628814697266, 0.027939558029174805, 0.06578516960144043, 0.35595083236694336, 10.475123167037964]


Quadratic Probing: Execution Time vs Table Size for Different Hash Functions
Quadratic Probing - Division Method
Quadratic Probing - Mid Square Method
Quadratic Probing - Folding Method
Quadratic Probing - Multiplication Method
Execution Time (seconds)
Table Size
1e7


Linear Probing: Execution Time vs Table Size for Different Hash Functions
Linear Probing - Division Method
Linear Probing - Mid Square Method
Linear Probing - Folding Method
Linear Probing - Multiplication Method
Execution Time (seconds)
Table Size
1e7





Observations:

Linear probing generally exhibits lower execution times compared to quadratic probing, double hashing, and chaining across different hash functions. Quadratic probing and double hashing show similar execution times, with quadratic probing sometimes slightly outperforming double hashing. Chaining tends to have higher execution times compared to the other methods, especially when the table size increases. As the table size increases, the execution times of all probing methods tend to increase, indicating that larger tables lead to more collisions. The scaling behavior differs slightly between the hash functions, such as the mid square

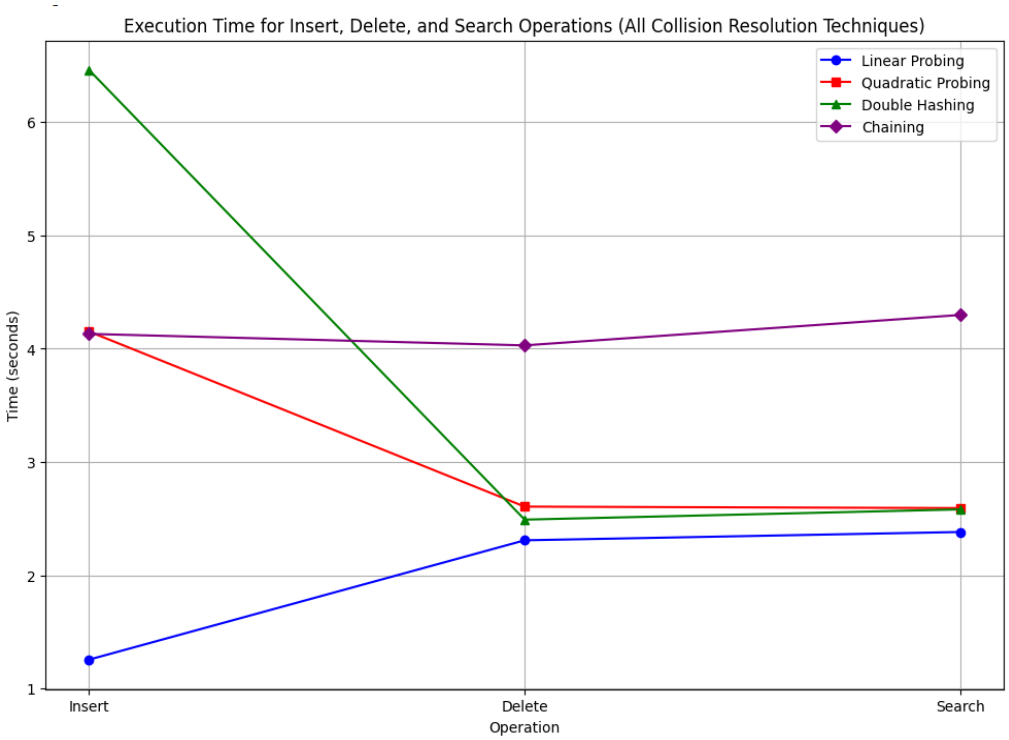
method and folding method, showing more consistent increases in execution time as the table size increases. Overall, the choice of hash function and probing method can have an impact on the performance of a hash table implementation, with Linear Probing generally performing well across different scenarios.

Experiment 3 : Comparison of collision resolution techniques in hash tables based on operations

The experiment compares the performance of four collision resolution techniques (Linear Probing, Quadratic Probing, Double Hashing, and Chaining) in a hash table implementation. The hash function used in the code is a simple division operation. Using a fixed table size of 10,000,000 and an input size of 1,000,000, the execution times for Insert, Delete, and Search operations are measured for each technique. Search and delete operations are performed on the inserted values.

Results:

Collision Resolution Technique	Insert Time	Delete Time	Search Time	Total Time
Linear Probing	1.25689	2.30983	2.38323	5.94996
Quadratic Probing	4.15196	2.60714	2.59418	9.35328
Double Hashing	6.46017	2.49224	2.58267	11.5351
Chaining	4.13149	4.03011	4.29834	12.4599



Observations:

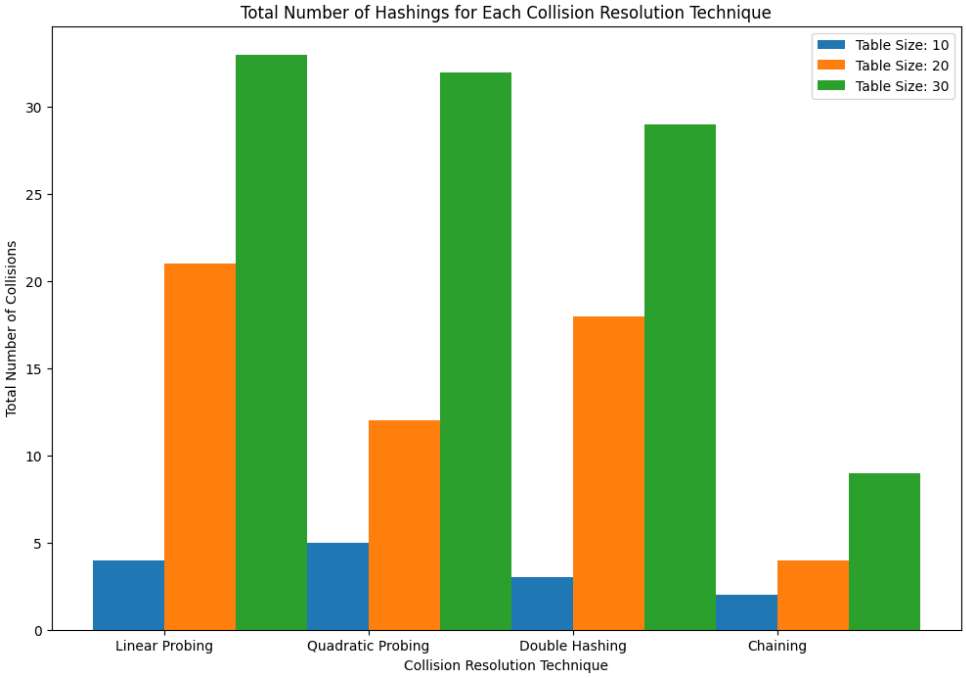
It can be observed that Chaining consistently outperforms the other collision resolution techniques in terms of total execution time, with the lowest times recorded for Insert, Delete, and Search operations. Quadratic Probing shows a balanced performance across operations, while Linear Probing tends to have higher execution times, especially in Delete operations. Double Hashing performs relatively well in Insert and Search operations but exhibits higher times for Delete operations. Overall, Chaining appears to be the most efficient technique for this experiment.

Experiment 4 : Analyzing collision rates in hash tables with different collision resolution techniques

This program simulates a hash table experiment with different collision resolution techniques (Linear Probing, Quadratic Probing, Double Hashing, and Chaining) and varying table sizes (10, 20, and 30). For each table size, a set of random values is generated, and each value is inserted into a hash table using the specified collision resolution technique. The hash function used in the code is a simple division operation. The total number of collisions encountered during the insertion process is recorded for each technique and table size combination. The load factor of the hash table after insertion is 0.75 for all table sizes.

Results:

```
Table Size: 10
Linear Probing:  Total Collisions: 4
Quadratic Probing:  Total Collisions: 5
Double Hashing:  Total Collisions: 3
Chaining:  Total Collisions: 2
Table Size: 20
Linear Probing:  Total Collisions: 21
Quadratic Probing:  Total Collisions: 12
Double Hashing:  Total Collisions: 18
Chaining:  Total Collisions: 4
Table Size: 30
Linear Probing:  Total Collisions: 33
Quadratic Probing:  Total Collisions: 32
Double Hashing:  Total Collisions: 29
Chaining:  Total Collisions: 9
```



Observation:

The total number of collisions varies with collision resolution technique and table size. Linear probing demonstrates a consistent increase in collisions with larger tables, indicating clustering issues. Quadratic probing has a fluctuating pattern, indicating some improvement over linear probing but still prone to clustering. Double hashing has relatively stable and lower collision rates, implying better performance in spreading out elements. Chaining offers good performance but shows moderate increases in collisions with larger tables. Overall, double hashing and chaining appear to be more effective at handling collisions than linear and quadratic probing, which struggle with clustering.

Experiment 5 : Comparing the performance of collision resolution techniques with varying load factors

The experiment varies the load factor (the ratio of the number of elements to the table size) to observe how each technique handles different levels of data density. For each load factor, the experiment measures the average execution time for inserting elements into the hash table using each collision resolution technique — linear probing, quadratic probing, double hashing, and chaining. The hash function used in the code is a division operation.

Results:

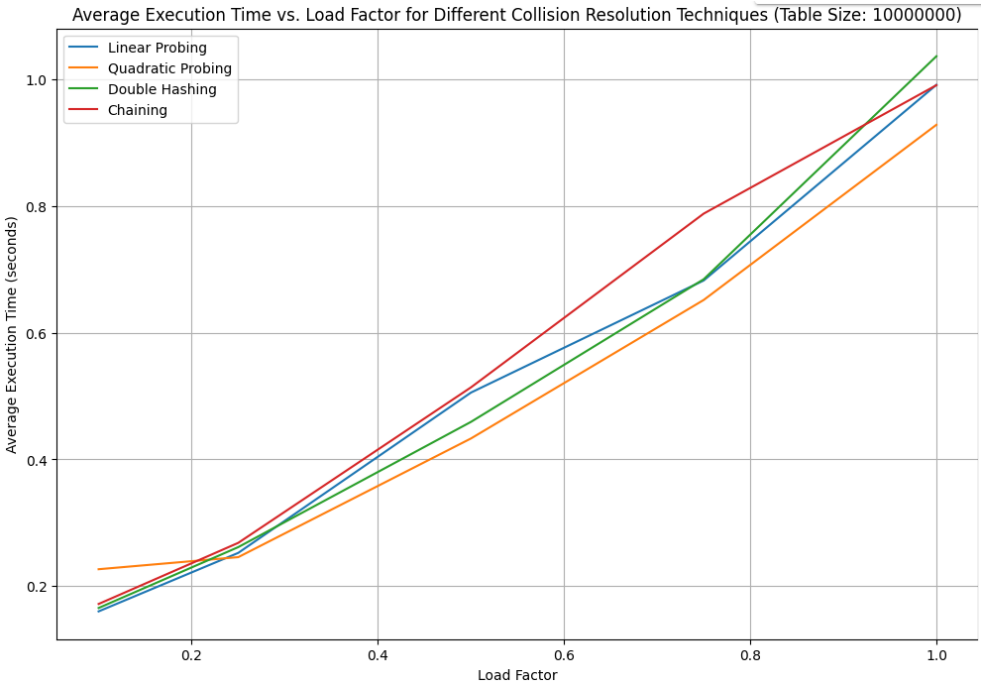
```
Input Size: 100000
Table Size: 1000000
Load factor: 0.1
Linear Probing: 0.1602299690246582 seconds
Quadratic Probing: 0.22673501968383789 seconds
Double Hashing: 0.16559290885925293 seconds
Chaining: 0.17191872596740723 seconds

Input Size: 250000
Table Size: 1000000
Load factor: 0.25
Linear Probing: 0.2525036811828613 seconds
Quadratic Probing: 0.2457890033721924 seconds
Double Hashing: 0.2616166114807129 seconds
Chaining: 0.2683344841003418 seconds

Input Size: 500000
Table Size: 1000000
Load factor: 0.5
Linear Probing: 0.5054683685302734 seconds
Quadratic Probing: 0.4330556392669678 seconds
Double Hashing: 0.45923490524291993 seconds
Chaining: 0.5137006759643554 seconds

Input Size: 750000
Table Size: 1000000
Load factor: 0.75
Linear Probing: 0.6823484420776367 seconds
Quadratic Probing: 0.6515814304351807 seconds
Double Hashing: 0.6843120098114014 seconds
Chaining: 0.7878387928009033 seconds

Input Size: 1000000
Table Size: 1000000
Load factor: 1.0
Linear Probing: 0.9910526275634766 seconds
Quadratic Probing: 0.9280255317687989 seconds
Double Hashing: 1.0361802577972412 seconds
Chaining: 0.9905174732208252 seconds
```



Observation:

The average execution time increases with the load factor due to more collisions. Linear Probing is less effective at higher load factors but advantageous at lower ones. Double Hashing and Quadratic Probing show stable performance, with Double Hashing performing slightly better due to its ability to prevent clustering. Chaining provides stable performance but typically executes slightly slower than probing techniques. Overall, Linear Probing is suitable for low to moderate load factors, while Double Hashing, Quadratic Probing, and Chaining offer stable performance across various load factors.

Conclusion

The report presents a comprehensive analysis of various collision resolution techniques in hash tables, including Linear Probing, Quadratic Probing, Double Hashing, and Chaining. Across multiple experiments, the performance of these techniques was evaluated based on factors such as load factor, table size, and operation types (insert, delete, search).

Overall, Linear Probing shows competitive performance at lower load factors but degrades as the load factor increases due to clustering. Quadratic Probing and Double Hashing exhibit more stable performance across different load factors and table sizes, with Double Hashing performing slightly better due to its ability to prevent clustering. Chaining provides consistent performance but tends to have slightly higher execution times compared to the probing techniques.

The choice of collision resolution technique and hashing algorithm depends on the specific requirements of the application. Each technique has its trade-offs in terms of execution times and efficiency, highlighting the importance of selecting the appropriate technique based on the specific characteristics of the application.

Citations:

[1] Educative, "Hash Table Collision Resolution." Educative: https://www.educative.io/answers/hash-table-collision-resolution

[2] UpGrad, "Hashing in Data Structure." UpGrad: https://www.upgrad.com/blog/hashing-in-data-structure/

[3] Indian Institute of Technology Kanpur, "Hash Tables: Hash Function, Collision Resolution Techniques, and Performance." Indian Institute of Technology Kanpur: https://ifacet.iitk.ac.in/knowledge-hub/data-structure-with-c/hash-tables-hash-function-collision-resolution-techniques-and-performance/

[4] GeeksforGeeks, "What is Hashing?" GeeksforGeeks: https://www.geeksforgeeks.org/what-is-hashing/

[5] LibreTexts, "Collision resolution." LibreTexts: https://eng.libretexts.org/Bookshelves/Computer_Science/Databases_and_Data_Structures/Book%3A_Data_Structures_(Wikibook)/09%3A_Hash_Tables/9.03%3A_Collision_resolution

[6] Enjoy Algorithm, "Introduction to Hashing in Programming," Medium, Oct. 11, 2021: https://medium.com/enjoy-algorithm/introduction-to-hashing-in-programming-617960aeccf2

[7] N. Karumanchi, Data Structures and Algorithms Made Easy: Data Structure and Algorithmic Puzzles, 2nd ed. CareerMonk Publications, 2011.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 4th ed. MIT Press, 2009.