Chapter 1: Hashing

By: Sandeep Godbole

In our fast-paced digital world, how we manage and retrieve data significantly impacts our experiences. Whether we're shopping online, browsing the web, or using applications, the need for quick and efficient data access is more important than ever.

Several techniques, such as indexing, binary search, bloom filters, and many more, have been developed for quick and efficient data access. Another important technique is hashing, which plays a key role in speeding up data retrieval. In this chapter, we will explore various aspects related to hashing.

Introduction to Hashing

Hashing is a technique used to map data to specific locations within a fixed-size table, known as a hash table. A hash function generates an index (hash value) for each input, allowing for efficient data storage, retrieval, and management. This process reduces search times to nearly constant time, especially in well-designed hash tables. Hashing is widely applied in scenarios such as database indexing, caching, and data retrieval in applications like dictionaries and password storage.

Vector versus Hash Table

Vectors and hash tables share a fundamental similarity: both use a dynamic array as their underlying storage. However, the way they organize and access data differs significantly. In a vector, there is no direct relationship between an element's index and its value, so finding a specific value requires examining each element individually, resulting in a time complexity of $O(n)$. In a hash table, a hash function establishes a relationship between each key and a specific index, allowing for direct access to values. This mapping enables hash tables to retrieve values in constant time, with a search time complexity of $O(1)$.

💡 A **key** is a unique identifier used to access a specific value or record within a data structure.

Components of a Hash Table

A hash table consists of several key components; however, for now, we will explore two of them: the array (or bucket) and the hash function.

- **Array**: The primary structure where data is stored.
- **Hash Function**: A function that takes a key as input and outputs an index (position in the array) where the corresponding value should be stored. The hash function aims to distribute keys evenly across the array to minimize collisions.
- **Buckets**: The structure used to store multiple values at the same index when a collision occurs.
- **Collision Resolution Method:** A strategy to handle cases where two keys hash to the same index.
- **Keys:** Unique identifiers that are hashed to determine where in the array to store their associated values.
- **Values**: Data associated with keys, stored at the array location indicated by the hash function.
- **Load Factor**: A measure that indicates how full the hash table is.
- **Resize/rehash mechanism**: A mechanism to increase the size of the hash table and rehash existing entries when the load factor crosses a certain threshold.

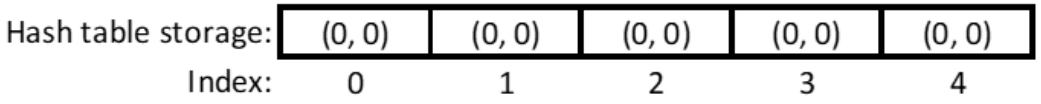
Illustration

A recent exam was conducted in which students from a classroom scored various marks. Each student has a unique identifier (student ID) and their corresponding score. Given the following data, create a hash table using a simple hash function that computes the index by taking the modules of the student ID with the size of the hash table (let's assume a size of 5).

Student ID	Score
100	85
101	90
102	78
103	92
104	88

Inserting Records into a Hash Table

We begin with a hash table containing 5 buckets and a simple hash function, `index = key % 5`, which computes the index by taking the modulus of the student ID with the size of the hash table.



Fig, 1: The hash table in its initial state.

To insert records into a hash table, start by passing the Student ID (the key) to the hash function, which calculates the index using the formula `index = key % 5`. For example, if the Student ID is 100 and the hash table size is 5, the index will be `index = 100 % 5 = 0`. Next, check if the calculated index is already occupied. If the bucket at this index is empty, insert the new record directly.

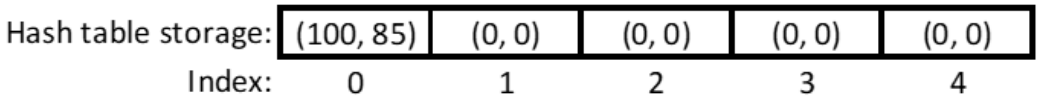


Fig. 2: The hash table after inserting the record (100, 85).

After repeating the same procedure for the remaining records, the hash table looks as follows:

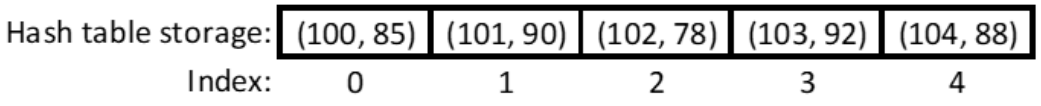


Fig. 3: The hash table after inserting all the records.

Searching a Record in a Hash Table

To search for or retrieve a record from a hash table, start by passing Student ID (the key) to the hash function to compute the corresponding index. Use the calculated index to access the appropriate bucket in the hash table. If a match is found, retrieve and return the associated value (e.g., the student's score). If no match is found, return an indication that the record does not exist.

For instance, to retrieve the record for Student ID 100:

- Compute the index using the hash function: `index = 100 % 5`, which gives you 0.
- Access the bucket at index 0 and check for the record.
- Return the record if found, or indicate it doesn't exist if not.

Since the execution of the above algorithm takes constant time, the time complexity for searching a record in a hash table is considered O(1). This contrasts with storing records in a simple vector, where the time complexity for searching is O(n).

Deleting a Record from a Hash Table

To delete a record from the hash table, first pass the Student ID (the key) to the hash function to compute the corresponding index. Use the calculated index to access the appropriate bucket in the hash table. Once the record is found, remove it from the bucket or mark it as deleted as appropriate.

For instance, if you want to delete the record for Student ID 101:

- Compute the index using the hash function: `index = 101 % 5`, which gives you 1.
- Access the bucket at index 1 and look for the record.
- Remove the record if found.

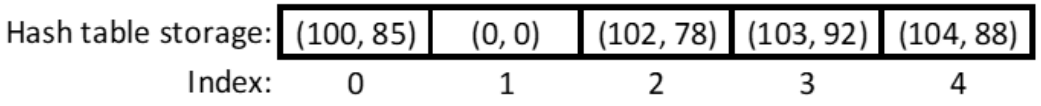


Fig. 4: The hash table after the deletion of the record (101, 90).

Collision in Hash Tables

A collision occurs when you attempt to insert a new record into a hash table, but the index calculated by the hash function is already occupied by a previously stored record. This situation arises due to the limited size of the

hash table and the nature of the hash function, which may produce the same index for different keys (also known as hash collisions).

For example, let's try to insert a new record for Student ID 106 into a hash table that is already filled with records for Student IDs 100 to 104.

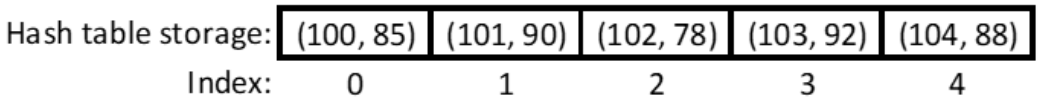


Fig. 5: The hash table after inserting all the records.

We begin calculating the index using the hash function: `index = 106 % 5 = 1`. The hash function indicates that the index for Student ID 106 is 1. Upon checking index 1, we find that it is already occupied by Student ID 101. This situation is known as a **collision**.

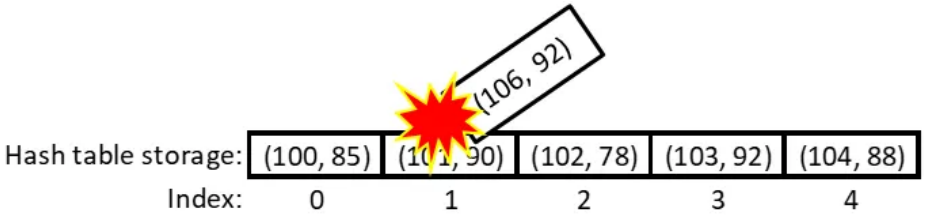


Fig. 6: The record (106, 92) is colliding with the record (101, 90).

A **collision** in a hash table occurs when two or more keys hash to the same index. This results in multiple values being stored at the same location.

Collision Resolution in Hash Table

Collision resolution is necessary in hash tables when multiple keys hash to the same index. There are two primary techniques: **closed addressing** and **open addressing**.

- **Closed Addressing**: In this technique, all colliding elements are stored in the same bucket, often using a data structure like a linked list or a dynamic array. Examples of closed addressing techniques include **chaining**.
- **Open Addressing**: In open addressing, when a collision occurs, the hash table itself searches for an alternative open slot within the table to store the

value. Examples of open addressing techniques include **linear probing**, **quadratic probing**, and **double hashing**.

> 💡 **Probing** is a technique used in hash tables to resolve collisions, which occur when multiple keys hash to the same index. The goal of probing is to find an alternative index within the hash table to store the colliding entry.

In the next section, we will explore each collision resolution technique in detail.

Chaining

In the chaining method of collision resolution, each slot in the hash table is associated with a linked list (or chain) that holds all keys hashing to that specific index. This way, if multiple keys have the same hash value, they are stored as nodes within the same list rather than occupying separate locations in the array. This approach keeps collisions contained within individual lists, making it both efficient and simple to manage.

Let's assume a hash table with 10 buckets and a modulo-10 hash function for inserting values 70, 22, 60, 8, 41, 72, 96, and 20. In this structure, each bucket initially points to an empty list, and we'll insert each key by determining its index and adding it to the linked list at that location.

Insertion Process

The process of inserting a key in a chaining hash table is straightforward: the hash function calculates the index, and if the bucket already contains nodes, the new key is added to the beginning (or end) of the linked list.

- **Initial State of Hash Table:** The hash table starts with 10 empty buckets. See Fig. 7(a).
- **Inserting 70:** The hash function `70 % 10` results in index 0. Since index 0 is empty, a new list is created at index 0, and 70 is inserted as the first node in this list. See Fig. 7(b).
- **Inserting 22:** The hash function `22 % 10` results in index 2, which is also empty. A new linked list is created at index 2, and 22 is added as the first node. See Fig. 7(c).

- **Inserting 60:** With `60 % 10`, we get index 0. A collision occurs here since index 0 already contains key 70. Using chaining, 60 is inserted at the beginning of the linked list at index 0, making the chain at index 0 contain keys [60, 70]. See Fig. 7(d).
- **Inserting 8:** The hash function `8 % 10` calculates index 8, which is empty. A new list is created, and 8 is added to this list at index 8. See Fig. 7(e).
- **Inserting 41:** The function `41 % 10` yields index 1, which is unoccupied. A new linked list is created at index 1, and 41 is inserted as the first node. See Fig. 7(f).
- **Inserting 72:** With `72 % 10`, we find index 2, where key 22 is already present. Using chaining, 72 is added to the list at index 2, which now contains [72, 22]. See Fig. 7(g).
- **Inserting 96:** The hash function `96 % 10` gives index 6. Since index 6 is empty, a new list is created, and 96 is inserted at this location. See Fig. 7(h).
- **Inserting 20:** The hash function `20 % 10` results in index 0. Since index 0 already contains [60, 70], key 20 is added to the beginning of the list at index 0. Now, index 0 holds [20, 60, 70] See Fig. 7(i).

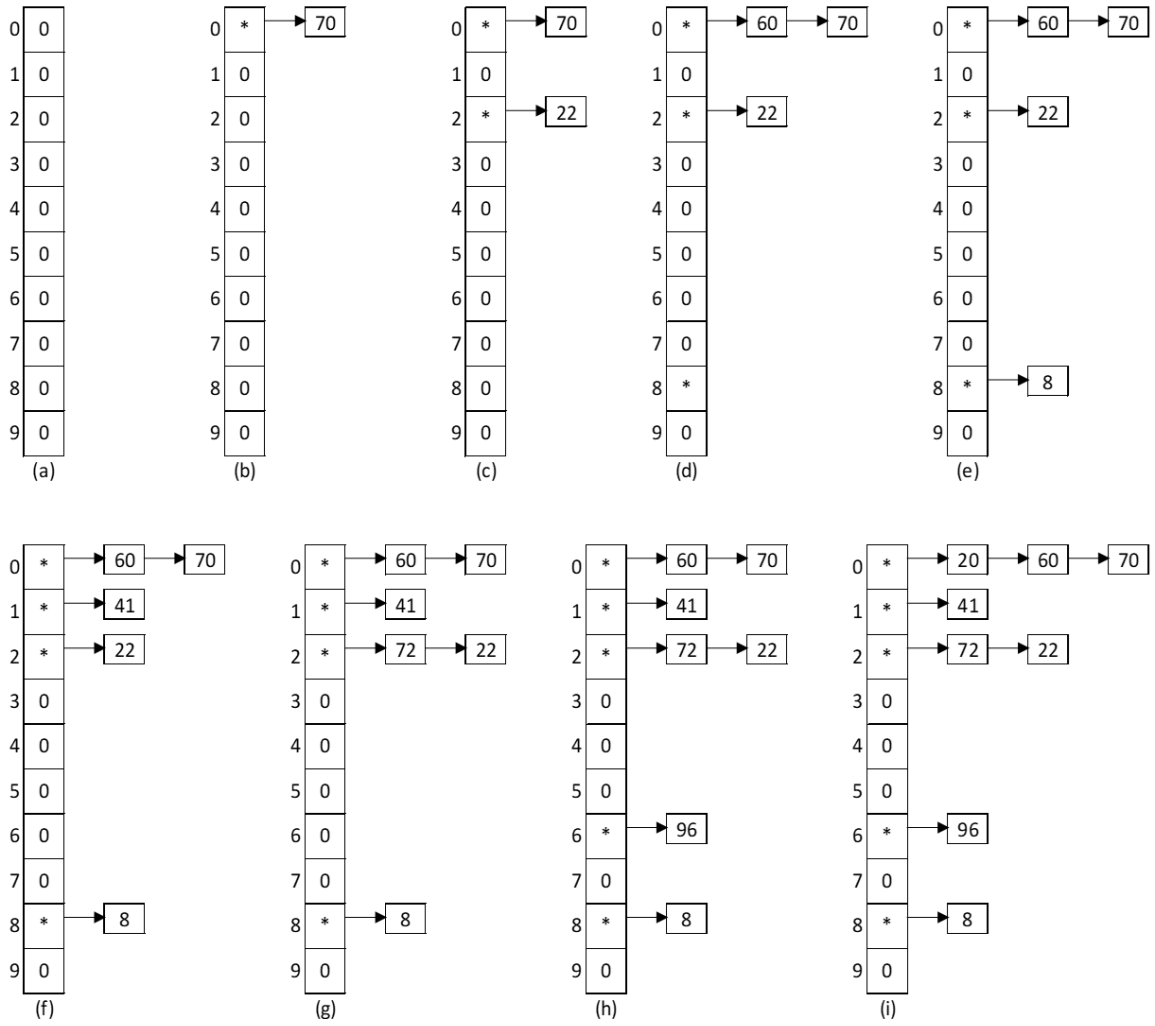


Fig. 7: The hash table after inserting all the records.

Each bucket with more than one node contains a chain, accommodating collisions effectively.

Searching Process

Searching in a chaining hash table involves computing the index using the hash function, navigating to that index, and traversing the linked list to locate the key. The search complexity is based on the length of the list at a given index.

- **Searching for Key 70:** The hash function `70 % 10` computes index 0. We then traverse the linked list at index 0, checking each node. Starting from the first node, we find that key 20 does not match, so we continue to key 60 and then to key 70, where a match is found. The associated data or record for key 70 is then retrieved.

Chaining offers a significant advantage in search efficiency, especially when the load factor (number of entries relative to the table size) remains low, as each linked list tends to remain short.

Deletion Process

To delete a key in a chaining-based hash table, we calculate its index, access the linked list at that index, and search for the key. If found, the node containing the key is removed from the list.

- **Deleting Key 70:** The hash function `70 % 10` gives index 0. At index 0, the linked list contains [20, 60, 70]. We traverse this list to find key 70. Upon locating it, we delete the node, and the list at index 0 becomes [20, 60].

Chaining allows for effective deletion without disrupting the structure of the table, as each list remains self-contained. If keys are frequently added and removed, chaining offers a flexible solution as it can easily manage varying numbers of keys without rehashing.

Summary of Chaining

Chaining is an efficient collision resolution strategy, especially useful when hash tables experience a high load factor. By keeping each chain self-contained, it avoids clustering and offers efficient insertion, search, and deletion operations, even under high occupancy conditions.

💡 **Time complexity of insertion, deletion and searching operations in chaining**

- **Insertion**: O(1) on average. Inserting a new element at the front of the list results in O(1) time complexity.
- **Deletion**: O(1) on average. Deletion involves finding the element in the list at the corresponding index and removing it, which is efficient if the list is short. However, it can degrade to O(n) if a large number of elements hash to the same value.
- **Searching**: O(1) on average if the bucket has only a few elements. In the worst case, if all elements hash to the same index, searching could degrade to O(n), where n is the number of elements in the hash table.

This distinction highlights the importance of having a good hash function and an appropriate load factor to maintain efficient access times.

Linear Probing

Linear probing is a collision resolution technique used in open addressing, where, upon encountering a collision, the hash table sequentially checks the next index until an empty slot is found. This technique simply increments the index by 1 each time, wrapping around the table if necessary. The primary goal of linear probing is to avoid collisions while maintaining a simple, contiguous probing sequence. However, it can sometimes lead to a phenomenon called *primary clustering,* where clusters of consecutive occupied slots make future collisions more likely in that cluster area.

Primary clustering is a phenomenon that occurs in hash tables using open addressing, particularly with linear probing. When multiple keys hash to the same index, they are resolved by sequentially probing the next available slots. Over time, this causes clusters of filled slots to form around certain indices, making future insertions and searches more likely to fall into these clusters, further extending them.

This leads to performance degradation, as the time to find an empty slot or to search for a key increases due to the growing size of these clusters. Even keys that do not hash to the same index but land near a cluster end up contributing to its growth, exacerbating the problem.

In essence, primary clustering is the concentration of filled hash table buckets into large, contiguous blocks, which can significantly slow down operations, making them less efficient than the expected constant time $O(1)$ for hash table access.

Let's consider a hash table with 10 buckets (indexed 0 through 9), where each bucket can store only one key. We'll use the hash function `index = key % 10` to insert the values 70, 22, 60, 8, 41, 72, 96, and 20.

Insertion Process

To insert each key, we calculate the initial index and, if a collision occurs, apply linear probing by checking the next sequential index.

- **Initial State of Hash Table:** The hash table starts with 10 empty buckets. See Fig. 8(a).
- **Inserting 70:** The hash function `70 % 10` results in index 0. Since index 0 is empty, we insert key 70 at this index without any issues. See Fig. 8(b).
- **Inserting 22:** The hash function `22 % 10` yields index 2. Index 2 is free, so key 22 is placed there. See Fig. 8(c).
- **Inserting 60:** Applying the hash function `60 % 10` results in index 0, but this index already holds key 70. Linear probing now checks index 1, which is empty, so we insert key 60 at index 1. See Fig. 8(d).
- **Inserting 8:** The hash function `8 % 10` yields index 8. As this index is empty, key 8 is inserted at index 8. See Fig. 8(e).

- **Inserting 41:** Using `41 % 10`, we get index 1, which is occupied by key 60. Linear probing moves to index 2, but it is occupied by key 22. Probing continues to index 3, which is free, so key 41 is inserted at index 3. See Fig. 8(f).
- **Inserting 72:** The hash function `72 % 10` results in index 2. As this index is already taken by key 22, linear probing directs us to index 3 (occupied by key 41), then to index 4, which is empty. Key 72 is placed at index 4. See Fig. 8(g).
- **Inserting 96:** The hash function `96 % 10` yields index 6, which is empty, so key 96 is inserted at this index. See Fig. 8(h).
- **Inserting 20:** The hash function `20 % 10` gives index 0, already occupied by key 70. Probing continues sequentially through indices 1 (key 60), 2 (key 22), 3 (key 41), and 4 (key 72). Finally, index 5 is found empty, so key 20 is placed there. See Fig. 8(i).

After inserting all keys, the table looks like this: index 0 (70), index 1 (60), index 2 (22), index 3 (41), index 4 (72), index 5 (20), index 6 (96), and index 8 (8).

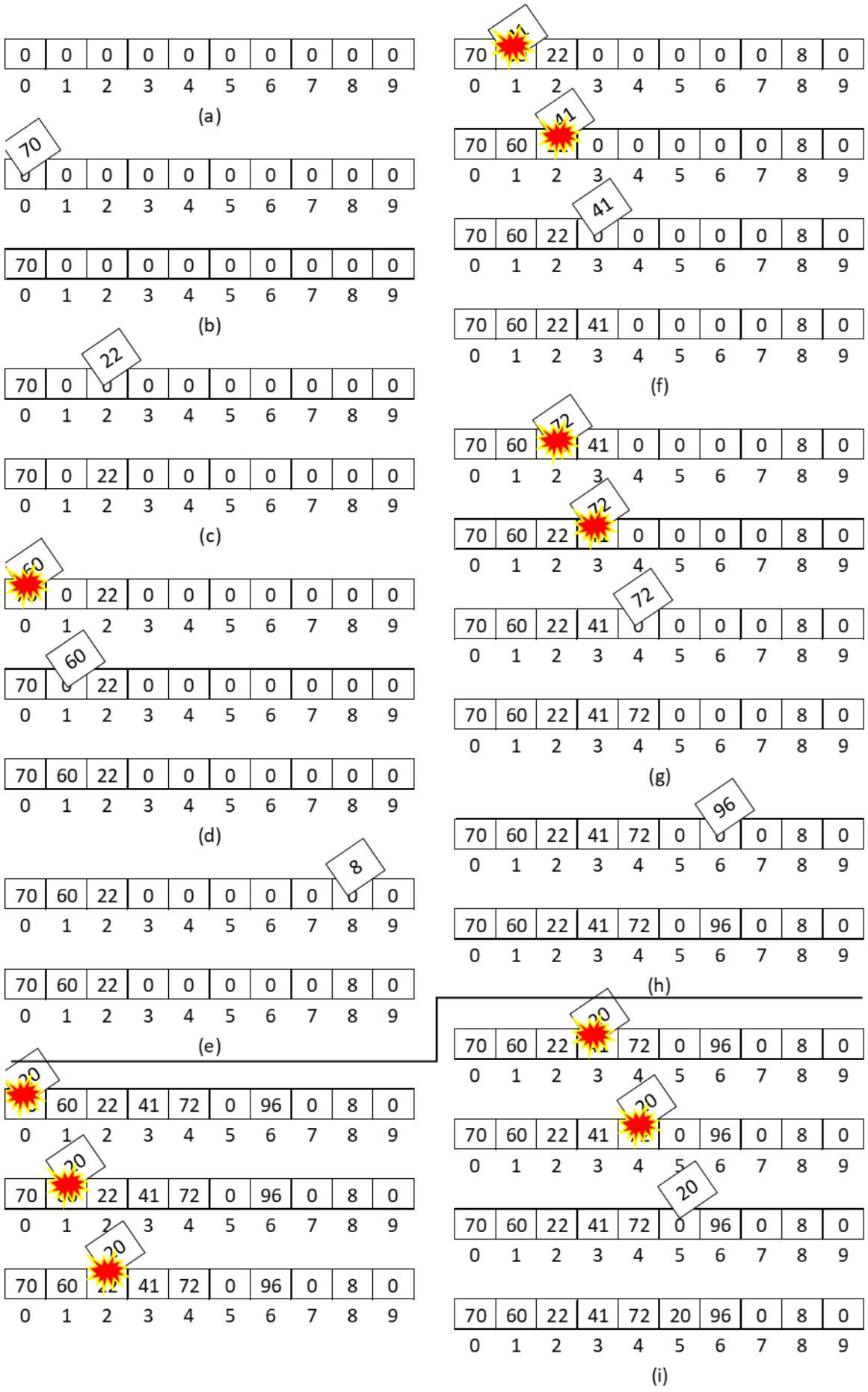

0 0 0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8 9
(a)
70
70 0 0 0 0 0 0 0 0 0
70 0 0 0 0 0 0 0 0 0
(b)
22
70 0 22 0 0 0 0 0 0 0
(c)
60
70 60 22 0 0 0 0 0 0 0
(d)
8
70 60 22 0 0 0 0 0 8 0
(e)
20
20 60 22 41 72 0 96 0 8 0
70 60 22 41 72 0 96 0 8 0
70 60 22 41 72 0 96 0 8 0
41
70 60 22 41 0 0 0 0 8 0
(f)
72
70 60 22 41 72 0 0 0 8 0
(g)
96
70 60 22 41 72 0 96 0 8 0
(h)
70 60 22 41 72 0 96 0 8 0
70 60 22 41 72 0 96 0 8 0
70 60 22 41 72 0 96 0 8 0
70 60 22 41 72 20 96 0 8 0
(i)

Fig. 8: The hash table after inserting all records using the linear probing strategy.

Searching Process

Searching in linear probing requires starting at the computed hash index and continuing linearly through the table until the key is found or an empty slot confirms the key is not present.

- **Searching for Key 20:** The hash function `20 % 10` initially directs us to index 0, which holds key 70. Linear probing then moves to index 1 (key 60), index 2 (key 22), index 3 (key 41), and index 4 (key 72). Finally, at index 5, key 20 is found. Thus, the search is successful after probing six positions.

Linear probing ensures that every key remains accessible even in the event of collisions, though searches may require multiple comparisons if clustering happens.

Deletion Process

In linear probing, deleting a key can disrupt the sequence of probes required for future searches. To maintain searchability, deleted slots are usually marked as “deleted” rather than simply emptied. This marker allows searches and insertions to continue probing past the deleted slot if needed.

- **Deleting Key 20:** The hash function `20 % 10` gives index 0, which contains key 70. Probing continues through indices 1, 2, 3, and 4, finally locating key 20 at index 5. The entry at index 5 is marked as deleted. This ensures that if a search for a different key probes index 5, the sequence continues as if it were merely an empty slot.

Linear probing is straightforward but can suffer from clustering, which can increase the time required for insertions, deletions, and searches as the table fills up.

💡 **Time complexity of insertion, deletion and searching operations in linear probing**

- **Insertion**: $O(1)$ on average, but can degrade to $O(n)$ in the worst case if the table is highly congested (when clustering occurs).
- **Deletion**: $O(1)$ on average, but can also degrade if rehashing or shifting elements is required due to clustering.
- **Searching**: $O(1)$ on average, but can degrade to $O(n)$ in the worst case due to clustering, as the search may need to scan through a long chain of occupied slots.

Quadratic Probing

Quadratic probing is a collision resolution technique in open addressing that resolves collisions by checking alternative positions in the hash table using a quadratic function. Instead of moving linearly to the next index (as in linear probing), it examines indices in a quadratic sequence from the original index generated by the hash function. For example, if a collision occurs at index `i`, quadratic probing will attempt to place the element at positions like `(i + 1^2) % table_size`, `(i + 2^2) % table_size`, `(i + 3^2) % table_size`, and so on. This spreads out the probes more widely and reduces the primary clustering issue common in linear probing.

Assume we have a hash table of 10 buckets and use a hash function `index = key % 10`. Let's go through the insertion, searching, and deletion processes step-by-step using the keys 70, 22, 60, 8, 41, 72, 96, and 20.

Insertion Process

To insert each key, we compute the initial hash index and then use quadratic probing if a collision occurs:

- **Initial State of Hash Table:** The hash table starts with 10 empty buckets. See Fig. 9(a).
- **Inserting 70:** The hash function `70 % 10` gives index 0. Since index 0 is empty, key 70 is placed at index 0. See Fig. 9(b)
- **Inserting 22:** The hash function `22 % 10` gives index 2. Since index 2 is unoccupied, key 22 is inserted at index 2. See Fig. 9(c).

- **Inserting 60:** The hash function `60 % 10` gives index 0, but index 0 is already occupied by key 70. Quadratic probing now examines index `(0 + 1^2) % 10 = 1`, which is empty, so key 60 is placed at index 1. See Fig. 9(d).
- **Inserting 8:** The hash function `8 % 10` gives index 8, and since index 8 is empty, key 8 is inserted there. See Fig. 9(e)
- **Inserting 41:** The hash function `41 % 10` results in index 1, which is already occupied by key 60. Quadratic probing attempts index `(1 + 1^2) % 10 = 2`, but index 2 is occupied by key 22. It then checks index `(1 + 2^2) % 10 = 5`, which is empty, so key 41 is placed at index 5. See Fig. 9(f).
- **Inserting 72:** The hash function `72 % 10` gives index 2, already filled by key 22. Quadratic probing checks `(2 + 1^2) % 10 = 3`, which is empty, so key 72 is placed at index 3. See Fig. 9(g).
- **Inserting 96:** The hash function `96 % 10` gives index 6. Since index 6 is free, key 96 is inserted there. See Fig. 9(h).
- **Inserting 20:** The hash function `20 % 10` results in index 0, occupied by key 70. Quadratic probing examines indices sequentially: `(0 + 1^2) % 10 = 1` (occupied by 60), `(0 + 2^2) % 10 = 4` (empty). Thus, key 20 is placed at index 4. See Fig. 9(i).

After inserting all keys, the table has values at indices: 0 (70), 1 (60), 2 (22), 3 (72), 4 (20), 5 (41), 6 (96), and 8 (8).

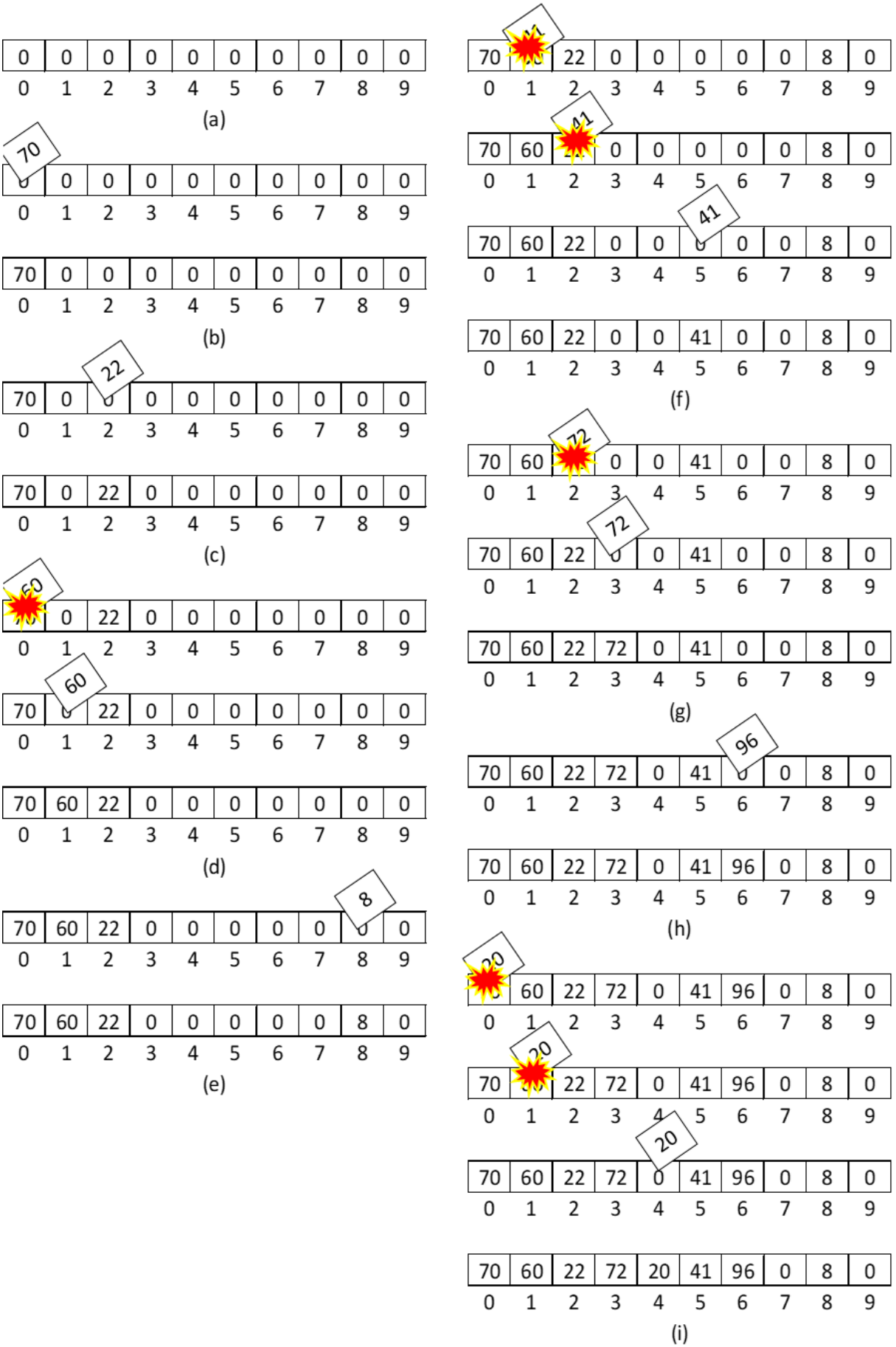

0 0 0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8 9
(a)
70
70 0 0 0 0 0 0 0 0 0
70 0 0 0 0 0 0 0 0 0
(b)
22
70 0 22 0 0 0 0 0 0 0
(c)
60
70 60 22 0 0 0 0 0 0 0
(d)
8
70 60 22 0 0 0 0 0 8 0
(e)
41
70 60 22 0 0 41 0 0 8 0
(f)
72
70 60 22 72 0 41 0 0 8 0
(g)
96
70 60 22 72 0 41 96 0 8 0
(h)
20
70 60 22 72 0 41 96 0 8 0
70 60 22 72 20 41 96 0 8 0
(i)

Fig. 9: The hash table after inserting all records using the quadratic probing strategy.

> 💡 **Secondary clustering** is a type of clustering in hash tables where keys that hash to the same index follow similar probe sequences, causing them to form localized clusters even if they started at different indices. This clustering effect is typically seen in quadratic probing and can degrade performance by increasing probe lengths, though it is less severe than primary clustering. Double hashing can mitigate secondary clustering by varying probe sequences for each key, creating more dispersed and efficient hash table distributions.

Searching Process

To search for a key using quadratic probing, the same sequence of indices is examined as would have been done during insertion:

- **Searching for Key 20:** The hash function `20 % 10` gives index 0, where key 70 is stored. Quadratic probing then checks `(0 + 1^2) % 10 = 1`, which contains key 60, and `(0 + 2^2) % 10 = 4`, where key 20 is found. Therefore, the search successfully retrieves key 20 after probing three locations.

The efficiency of quadratic probing is shown here, as it avoids clustering and allows rapid access to the target key even when collisions occur.

Deletion Process

Deleting a key with quadratic probing requires marking the bucket as deleted to preserve the probing sequence for future searches. This ensures that later searches do not stop prematurely.

- **Deleting Key 20:** First, the hash function `20 % 10` gives index 0, which contains key 70. Quadratic probing sequentially checks `(0 + 1^2) % 10 = 1` (key 60), and `(0 + 2^2) % 10 = 4`, where key 20 is found. Key 20 is then removed, leaving a marker at index 4 indicating that the spot is available but was previously occupied. This marker allows future searches that might have probed this index to continue accurately.

Quadratic probing minimizes clustering by widening the search path for empty indices. This is a beneficial approach for hash tables, especially when a table load factor is moderate, allowing efficient insertions, deletions, and retrievals without primary clustering problems.

💡 **Time complexity of insertion, deletion and searching operations in quadratic probing**

- **Insertion**: $O(1)$ on average, but can degrade to $O(n)$ if the table becomes heavily loaded or clustering occurs. The quadratic probing function helps reduce clustering, but it does not eliminate it entirely.
- **Deletion**: $O(1)$ on average, but similar to linear probing, it can require additional effort to handle deleted slots and maintain the integrity of the probing sequence.
- **Searching**: $O(1)$ on average, but can degrade to $O(n)$ in the worst case, especially if the table is near full and there are many collisions to resolve.

Double Hashing

Double hashing is an open addressing collision resolution technique that improves on linear probing and quadratic probing by minimizing clustering. In double hashing, when a collision occurs, we use a second hash function to determine the probe sequence. The probing distance (or step size) is dynamically calculated based on the second hash function, creating a more distributed probe sequence and reducing the chance of primary and secondary clustering.

Let's consider a hash table with 10 buckets (indexed from 0 to 9), where each bucket can hold only one key. We will use two hash functions for double hashing:

1. **Primary hash function:** `h1(key) = key % 10`
2. **Secondary hash function:** `h2(key) = 7 - (key % 7)`

The second hash function calculates a step size unique to each key, creating a sequence that skips over indices, reducing the clustering issue. Now, let's walk through the insertion, searching, and deletion processes using the example keys 70, 22, 60, 8, 41, 72, 96, and 20. We'll assume we want to retrieve and delete the record with key 20.

Insertion Process

For each key, we calculate the initial index using the primary hash function. If a collision occurs, we use the secondary hash function to determine the step size and probe forward.

- **Initial State of Hash Table:** The hash table starts with 10 empty buckets. See Fig. 10(a).
- **Inserting 70:** The primary hash function `h1(70) = 70 % 10` gives index 0. As index 0 is empty, key 70 is placed there. See Fig. 10(b).
- **Inserting 22:** The primary hash function `h1(22) = 22 % 10` gives index 2. Since index 2 is unoccupied, key 22 is inserted at index 2. See Fig. 10(c).
- **Inserting 60:** The primary hash function `h1(60) = 60 % 10` results in index 0, which is already occupied by key 70, causing a collision. The secondary hash function `h2(60) = 7 - (60 % 7) = 7 - 4 = 3` gives a step size of 3. We probe index `(0 + 3) % 10 = 3`, which is empty, so key 60 is inserted at index 3. See Fig. 10(d).
- **Inserting 8:** Using `h1(8) = 8 % 10`, we find index 8. As this index is empty, key 8 is placed there. See Fig. 10(e).
- **Inserting 41:** The primary hash function `h1(41) = 41 % 10` gives index 1, which is free. Therefore, key 41 is inserted at index 1. See Fig. 10(f).
- **Inserting 72:** The primary hash function `h1(72) = 72 % 10` yields index 2, which is occupied by key 22. The secondary hash function `h2(72) = 7 - (72 % 7) = 7 - 2 = 5` gives a step size of 5. Probing `(2 + 5) % 10 = 7`, we find index 7 is empty, so key 72 is inserted there. See Fig. 10(g).
- **Inserting 96:** The primary hash function `h1(96) = 96 % 10` gives index 6, which is free. Key 96 is placed at index 6. See Fig. 10(h).
- **Inserting 20:** The primary hash function `h1(20) = 20 % 10` yields index 0, which is occupied by key 70. The secondary hash function `h2(20) = 7 - (20 % 7) = 7 - 6 = 1` gives a step size of 1. Probing `(0 + 1) % 10 = 1`, index 1 is also occupied by key 41. Probing further `(0 + 2) % 10 = 2` and `(0 + 3) % 10 = 3` are both full. Finally, probing `(0 + 4) % 10 = 4` finds index 4 empty. Therefore, key 20 is inserted at index 4. See Fig. 10(i).

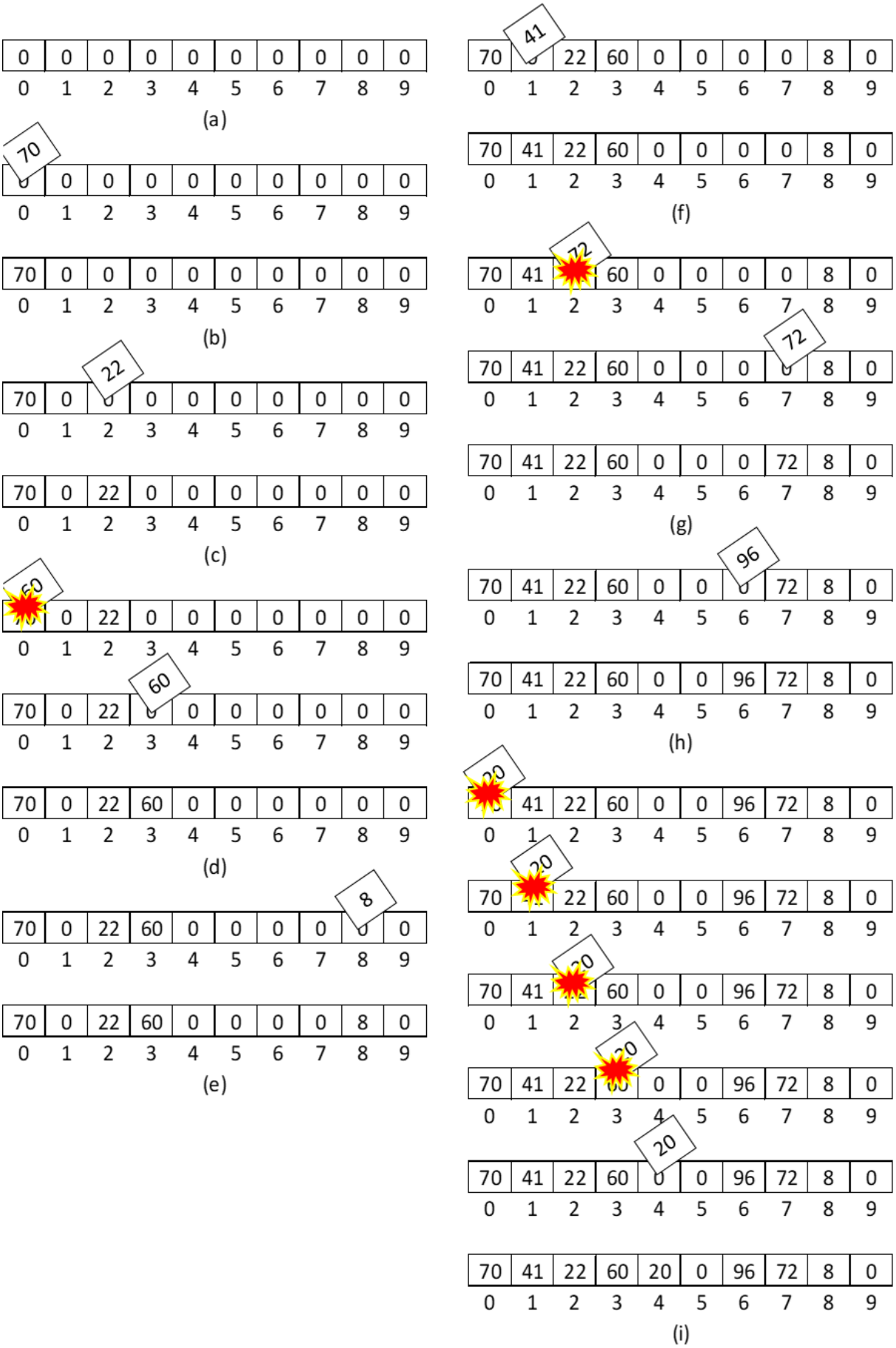

0 0 0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8 9
(a)
70
70 0 0 0 0 0 0 0 0 0
70 0 0 0 0 0 0 0 0 0
(b)
22
70 0 22 0 0 0 0 0 0 0
70 0 22 0 0 0 0 0 0 0
(c)
60
70 0 22 0 0 0 0 0 0 0
70 0 22 60 0 0 0 0 0 0
(d)
8
70 0 22 60 0 0 0 0 8 0
70 0 22 60 0 0 0 0 8 0
(e)
41
70 41 22 60 0 0 0 0 8 0
70 41 22 60 0 0 0 0 8 0
(f)
72
70 41 22 60 0 0 0 72 8 0
70 41 22 60 0 0 0 72 8 0
(g)
96
70 41 22 60 0 0 96 72 8 0
70 41 22 60 0 0 96 72 8 0
(h)
20
70 41 22 60 0 0 96 72 8 0
20
70 41 22 60 0 0 96 72 8 0
20
70 41 22 60 0 0 96 72 8 0
20
70 41 22 60 0 0 96 72 8 0
20
70 41 22 60 20 0 96 72 8 0
(i)

Fig. 10: The hash table after inserting all records using the double hashing strategy.

Searching Process

To search for a key, we calculate the primary index using the hash function and then use the second hash function to probe if necessary.

- **Searching for Key 20:** `h1(20) = 0` directs us to index 0, which holds key 70. Since it does not match, we use `h2(20) = 1` to continue probing by incrementing with a step size of 1 until we locate key 20 or exhaust the table.

Deletion Process

- **Deleting Key 20:** Deleting a key in double hashing follows a similar procedure as in searching. We locate the key using double hashing's probing sequence, then mark it as deleted (often by placing a sentinel value). If we want to delete key 20, we find it by applying the secondary probe sequence and then mark the location as deleted.

Double hashing ensures effective collision resolution and efficient access in hash tables while minimizing clustering effects.

> 💡 **Time complexity of insertion, deletion and searching operations in double hashing**
>
> - **Insertion**: O(1) on average, and provides better performance than linear or quadratic probing in handling collisions because it uses a second hash function to determine the next probe. However, it can still degrade to O(n) in the worst case when the table is very full.
> - **Deletion**: O(1) on average, with the potential for degradation if multiple probes are needed to find the deleted element.
> - **Searching**: O(1) on average and is generally more efficient than linear and quadratic probing because the second hash function reduces clustering. In the worst case, however, it can still take O(n) time.

What is Load Factor?

The load factor in hashing is a measure of how full a hash table is. It is calculated as the ratio of the number of elements (keys) currently stored in the hash table to the total number of available slots or buckets. Mathematically, it is expressed as $\alpha = n/m$, where n is the number of elements and m is the number of slots. A higher load factor increases the likelihood of collisions, which can degrade performance, prompting a resizing or rehashing of the hash table.

For example, the load factor of the following hash table is 0.8.

70	41	22	60	20	0	96	72	8	0
0	1	2	3	4	5	6	7	8	9

Rehashing

Rehashing is the process of resizing a hash table and redistributing its contents to reduce the load factor and improve performance. When a hash table becomes overly full, indicated by a high load factor, collisions increase, making insertion and search operations slower. During rehashing, a new, larger hash table is created, often with double the size of the current table. Each element in the original table is reinserted into the new table based on a new hash function or recalculated index, which distributes keys more evenly across the increased number of slots. Rehashing typically occurs automatically in implementations like `std::unordered_map` or `std::unordered_set` in C++.

What Are the Expectations from a Hash Function?

A hash function is expected to **distribute keys uniformly** across the hash table to minimize collisions, **be computationally efficient**, and **produce the same output for the same input each time**.

Various Hashing Algorithms

1. **Direct Method**: The key itself is used as the index in the hash table. This method is only practical when key values are small or restricted to a manageable range. For example, if keys are student roll numbers in a range from 1 to 100, then roll number 45 would map directly to index 45.
2. **Subtraction Method**: In this method, a constant value is subtracted from the key to compute the hash. This technique is simple but requires careful

selection of the constant to prevent clustering. For instance, if key 145 has 100 subtracted, the resulting hash index is 45.

3. **Digit Extraction Method**: Specific digits from the key are extracted and used to compute the hash. For example, in a 6-digit key like 123456, one could extract the 2nd and 5th digits to create an index like 25. This method works well when certain digits are uniformly distributed.
4. **Mid-Square Method**: This method squares the key and extracts a portion of the resulting value to serve as the index. For example, squaring key 56 gives 3136, and taking the middle digits (13) results in index 13. The mid-square method reduces clustering and works well with non-uniform key distributions.
5. **Folding Method**: This method breaks the key into equal-sized parts, often by digits, and adds them to produce the hash index. For example, key 432568 can be split into 432 and 568, which sum to produce index 1000. Folding is particularly useful for large numeric keys and helps spread the keys more evenly across the table.

std::unordered_set in C++

The `std::unordered_set` in C++ is an unordered associative container that stores unique keys, providing fast insertion, deletion, and search operations by using hashing. It is part of the Standard Template Library (STL) and provides an implementation for hash-based sets.

- **Insertion**: Elements are added using the `insert()` method. For example, `set.insert(42);` adds the element `42` to the unordered set.
- **Successful Search**: To search for an element, the `find()` method is used. If `set.find(42) != set.end()`, then the element `42` is present in the set.
- **Unsuccessful Search**: If `find()` returns `set.end()`, it indicates that the element is not in the set.
- **Removal**: Elements can be removed using the `erase()` method. Calling `set.erase(42);` removes the element `42` if it exists in the set.

The `std::unordered_set` automatically rehashes when the load factor becomes too high, ensuring performance consistency.

```
#include <iostream>
#include <unordered_set>
```

```
int main() {
    // Create an unordered_set of integers
    std::unordered_set<int> uset;

    // Insert elements into the unordered_set
    uset.insert(10);
    uset.insert(20);
    uset.insert(30);
    uset.insert(40);

    // Display elements in the unordered_set
    std::cout << "Elements in unordered_set: ";
    for (int elem : uset) {
        std::cout << elem << " ";
    }
    std::cout << std::endl;

    // Successful search
    int searchElement = 20;
    if (uset.find(searchElement) != uset.end()) {
        std::cout << "Element " << searchElement
            << " found in unordered_set." << std::endl;
    }
    else {
        std::cout << "Element " << searchElement
            << " not found in unordered_set." << std::endl;
    }

    // Unsuccessful search
    searchElement = 50;
    if (uset.find(searchElement) != uset.end()) {
        std::cout << "Element " << searchElement
            << " found in unordered_set." << std::endl;
    }
    else {
        std::cout << "Element " << searchElement
            << " not found in unordered_set." << std::endl;
```

```
    }

    // Remove an element
    uset.erase(20);
    std::cout << "After removing 20, elements in unordered_se
    for (int elem : uset) {
        std::cout << elem << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

std::unordered_map in C++

The `std::unordered_map` in C++ is an unordered associative container that stores key-value pairs, providing efficient access and management based on unique keys. Each key in the unordered map is unique and can be accessed in constant time, making it an ideal choice for implementing hash tables.

- **Insertion**: Key-value pairs are added using the `insert()` method or subscript operator. For instance, `map[42] = "example";` inserts a key `42` with a corresponding value `"example"`.
- **Successful Search**: The `find()` method is used to check if a key exists. If `map.find(42) != map.end()`, then key `42` is present in the map.
- **Unsuccessful Search**: If `find()` returns `map.end()`, it indicates that the key does not exist in the map.
- **Removal**: To remove an element, `erase()` is used. For example, `map.erase(42);` removes the key-value pair associated with key `42`.

The `std::unordered_map` dynamically manages its size by rehashing to maintain performance as elements are added.

```
#include <iostream>
#include <unordered_map>
#include <string>

int main() {
```

```
    // Create an unordered_map that maps strings to integers
    std::unordered_map<std::string, int> umap;

    // Insert key-value pairs into the unordered_map
    umap["Alice"] = 25;
    umap["Bob"] = 30;
    umap["Charlie"] = 35;

    // Display key-value pairs in the unordered_map
    std::cout << "Contents of unordered_map:" << std::endl;
    for (const auto& pair : umap) {
        std::cout << pair.first << ": " << pair.second << std
    }

    // Successful search
    std::string searchKey = "Alice";
    if (umap.find(searchKey) != umap.end()) {
        std::cout << "Key \"" << searchKey
            << "\" found with value " << umap[searchKey] << "
    }
    else {
        std::cout << "Key \"" << searchKey
            << "\" not found in unordered_map." << std::endl;
    }

    // Unsuccessful search
    searchKey = "David";
    if (umap.find(searchKey) != umap.end()) {
        std::cout << "Key \"" << searchKey
            << "\" found with value " << umap[searchKey] << "
    }
    else {
        std::cout << "Key \"" << searchKey
            << "\" not found in unordered_map." << std::endl;
    }

    // Remove a key-value pair
    umap.erase("Bob");
```

```
    std::cout << "After removing \"Bob\", contents of unorder
    for (const auto& pair : umap) {
        std::cout << pair.first << ": " << pair.second << std
    }

    return 0;
}
```

Applications of Hashing and Hash Tables

A Caching System

A caching system temporarily stores copies of frequently accessed data, allowing for rapid retrieval and reducing the load on slower storage systems. This technique is widely used in web applications, databases, and content delivery networks, resulting in faster performance and enhanced user satisfaction.

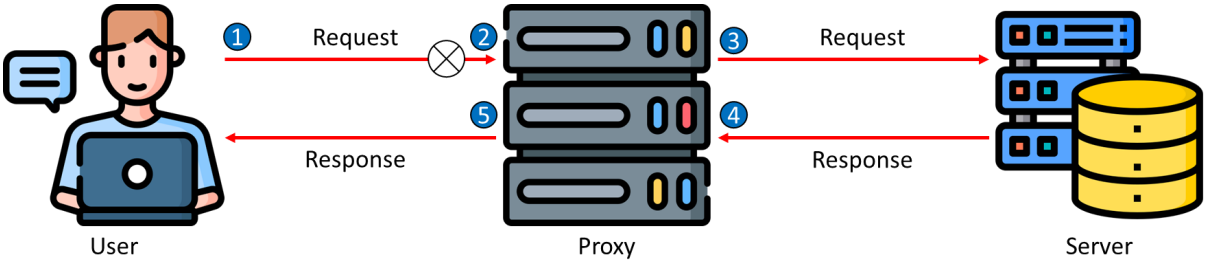


Fig. 11: Workflow when a cache miss occurs.

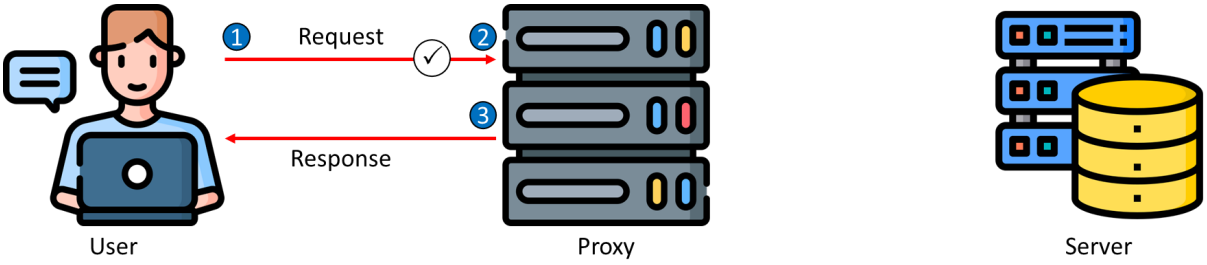


Fig. 12: Workflow when a cache hit occurs.

A Shopping Cart

An online shopping cart is essential for e-commerce platforms, enabling users to select and manage products before making a purchase. It needs to provide quick access to product details like prices and availability, ensuring a smooth

and enjoyable shopping experience as customers navigate through their choices.

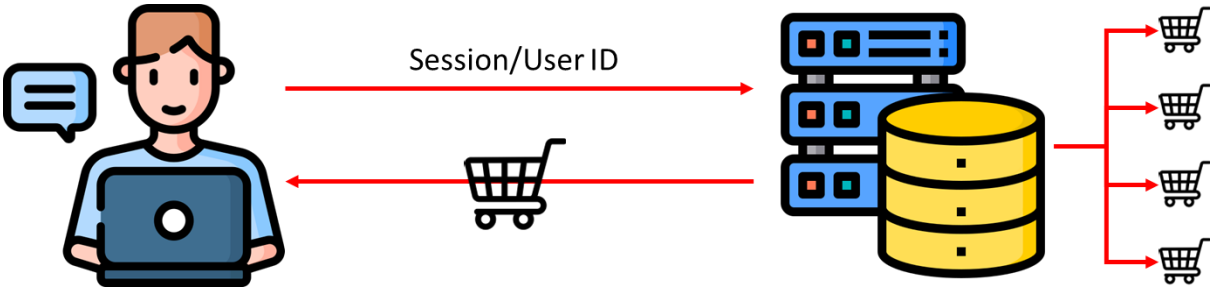


Fig. 13: A shopping cart workflow.

A DNS Resolver

DNS resolution translates human-readable domain names into IP addresses, allowing users to access websites effortlessly. This system must respond quickly to domain queries, facilitating efficient browsing and connectivity. When a user types a domain name, the DNS resolution process ensures the corresponding IP address is retrieved promptly, enabling seamless communication across the internet.

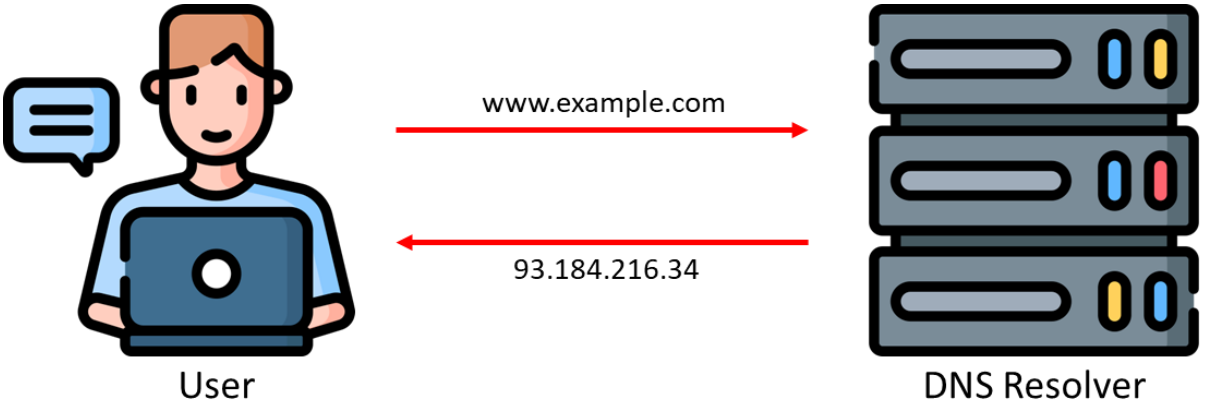


Fig. 14: A DNS resolver workflow.

These systems—a caching system, a shopping cart, and DNS resolution—highlight the crucial need for efficient data management in our digital interactions. At the heart of these systems is a powerful data structure known as a hash table, which enhances performance and streamlines data access.