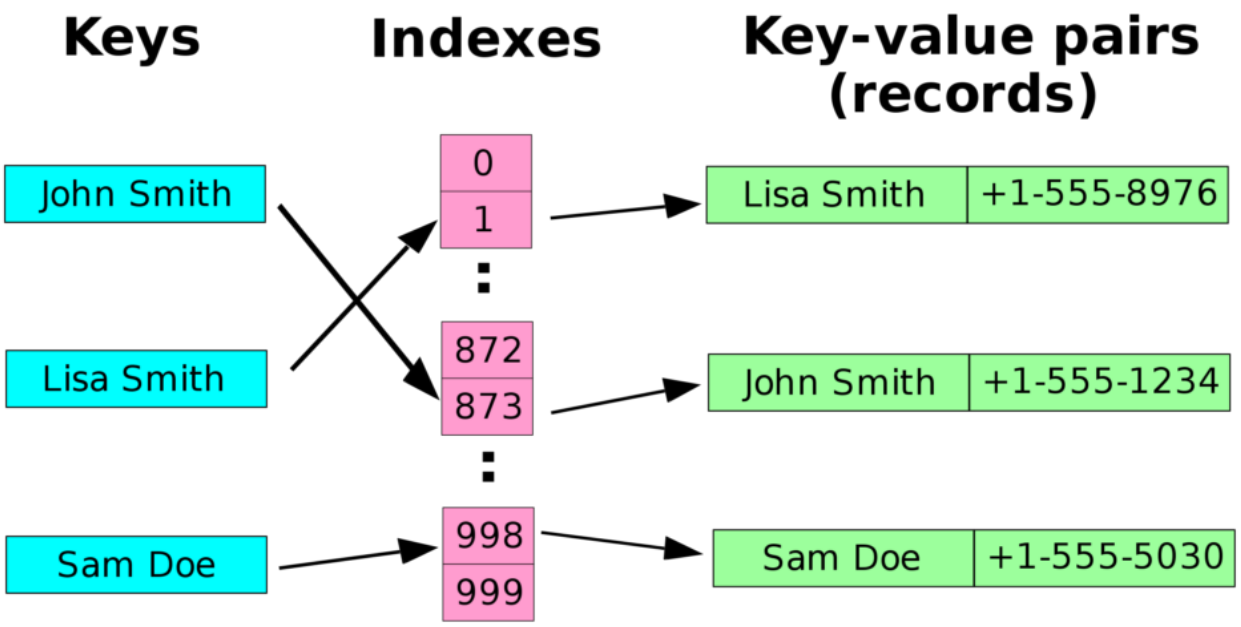
Session 12: Hash Functions and Hash Tables

In data structures, hashing is a technique used to map data elements to specific locations in an array, known as hash tables. Hash tables are data structures that provide fast access to data by using a hash function to compute an index or location for each element in the array. Hashing is widely used for data retrieval, indexing, and implementing associative arrays or dictionaries.



Introduction to Hash Tables:

A hash table is a data structure that allows efficient data retrieval based on key-value pairs. It is composed of an array (often referred to as the "bucket" or "table") and a hash function. The hash function takes a key as input and converts it into an integer, which is then used as an index to store the corresponding value in the array.

The key features of a hash table are:

- **Key-Value Pairs:** Each element in the hash table is stored as a key-value pair. The key is used as input to the hash function, which generates an index in the array where the value is stored. When searching for a value, the hash function is applied to the search key, and the corresponding index is used to retrieve the value in constant time (assuming a good hash function).
- **Hash Function:** The hash function is responsible for computing the hash code (or hash value) for each key. It should be deterministic, i.e., for the same key, it should always produce the same hash code. Additionally, it should aim to distribute the keys uniformly across the array to minimise collisions.

- **Collision Handling:** Collisions occur when two different keys produce the same hash code and attempt to be stored at the same index in the array. Proper collision handling mechanisms are essential to ensure that both keys and their associated values can be stored and retrieved accurately. Common collision resolution techniques include chaining (using linked lists) or open addressing (probing).
- **Load Factor**: The load factor of a hash table is the ratio of the number of stored elements (keys) to the total number of buckets (array size). A high load factor can result in increased collisions and reduced performance. To maintain efficient operations, it is essential to resize the hash table and rehash elements when the load factor exceeds a certain threshold.

Hash tables are known for their average-case constant-time (O(1)) complexity for insertion, deletion, and retrieval operations when the hash function distributes keys uniformly. However, in the worst case, when collisions are frequent, these operations may degrade to O(n), making it essential to choose a suitable hash function and handle collisions effectively.

Hash tables find extensive applications in programming languages, databases, compiler symbol tables, associative arrays, caches, and various other data-driven applications. They provide efficient access to data, making them a fundamental and valuable data structure in computer science.

Hash Functions

Hash functions and hash tables are two fundamental components used in data structures to efficiently store and retrieve data based on key-value pairs. Let's discuss hash functions first and then explore the different types of hash tables:

Hash Functions:

- Hash functions are mathematical algorithms that take an input (or 'message') of any length and produce a fixed-size string of characters, known as the hash value or hash code. The primary purpose of hash functions is to map data of arbitrary size to a fixed-size value, typically used for various applications, including data indexing, data integrity verification, and password storage.
- Characteristics of Hash Functions: As mentioned earlier, hash functions should be deterministic, produce fixed-size outputs, have fast computation, and exhibit the avalanche effect (small changes in input result in drastically different hash values).

Collision Resolution

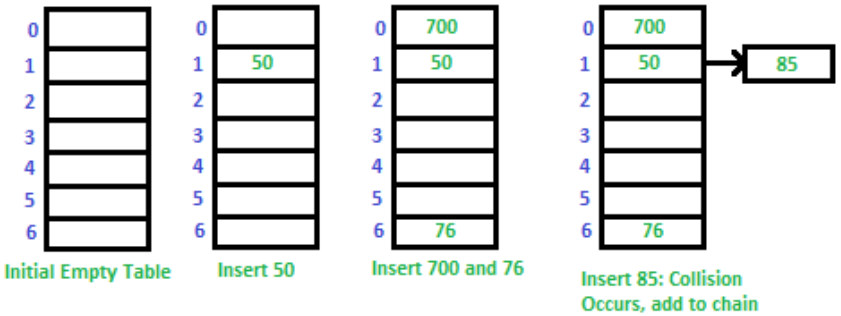
Collision resolution is a critical aspect of hash-based data structures, especially in hash tables, where collisions occur when two different keys produce the same hash code and attempt to be stored at the same index in the underlying array. There are various collision resolution techniques used to handle these collisions effectively.

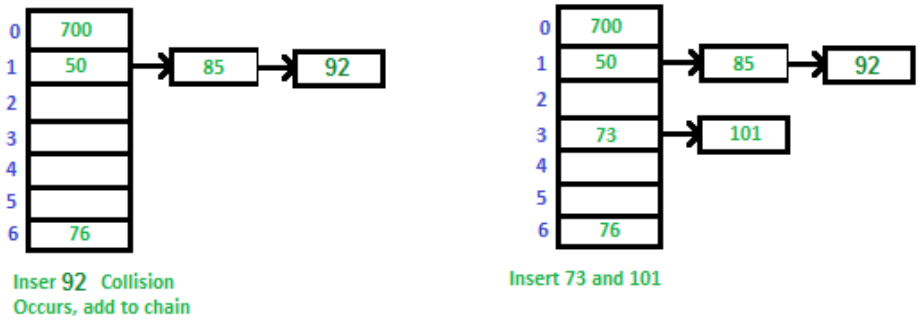
Index	Hashed keys
0	
1	
2	
3	123, 124, 135 (collision)
4	1267, 2378, 9087 (collision)
5	
6	
7	
8	
9	

Let's explore some common collision resolution methods:

Chaining (Separate Chaining):

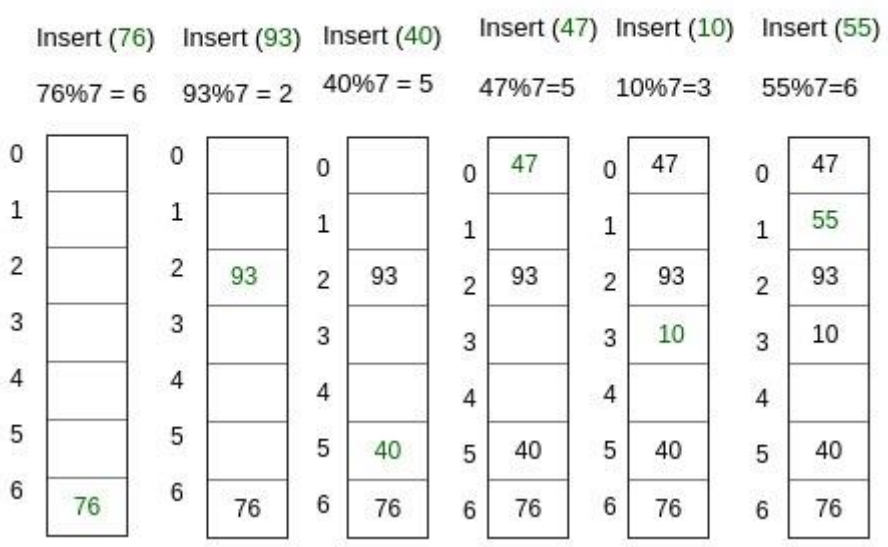
- Chaining is a popular collision resolution technique that involves using linked lists or other data structures to store multiple key-value pairs that share the same hash code.
- Each bucket in the hash table contains a linked list (or other data structure) to accommodate multiple elements with the same hash code.
- When a collision occurs, the new key-value pair is simply added to the linked list at the corresponding bucket, rather than overwriting the existing value. This way, multiple elements with the same hash code can coexist in the same bucket.





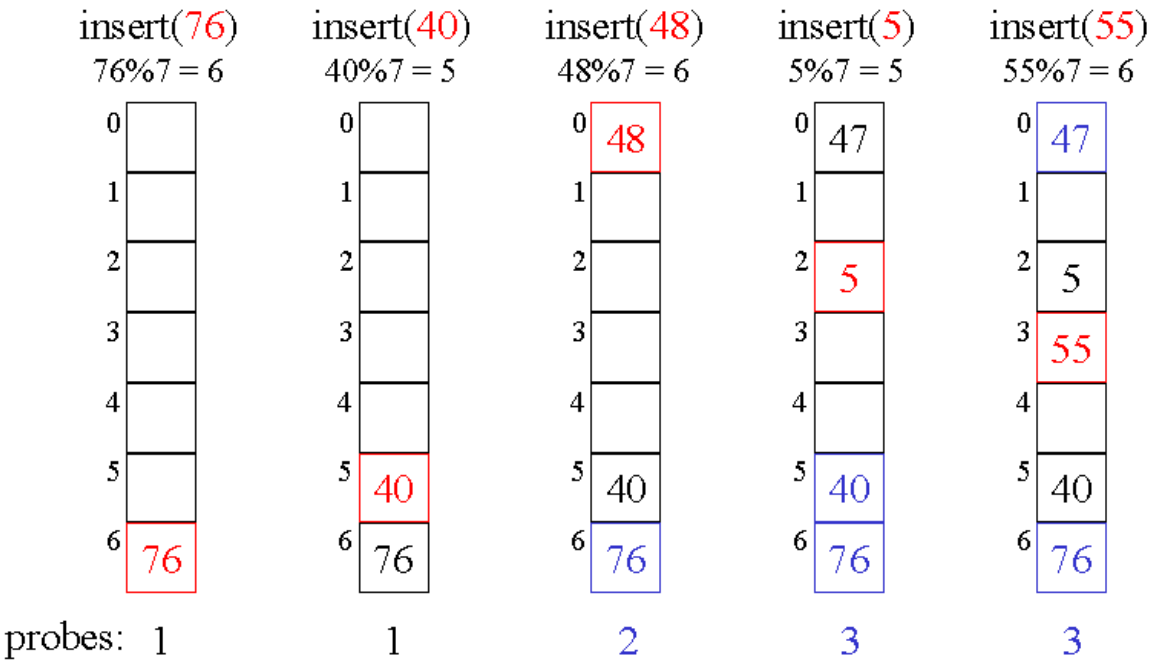
Linear Probing:

- Linear probing is a simple open addressing technique. When a collision occurs, the algorithm checks the next consecutive bucket in the array until an empty bucket is found.
- If the next bucket is occupied, it continues checking the subsequent buckets until an open slot is found.



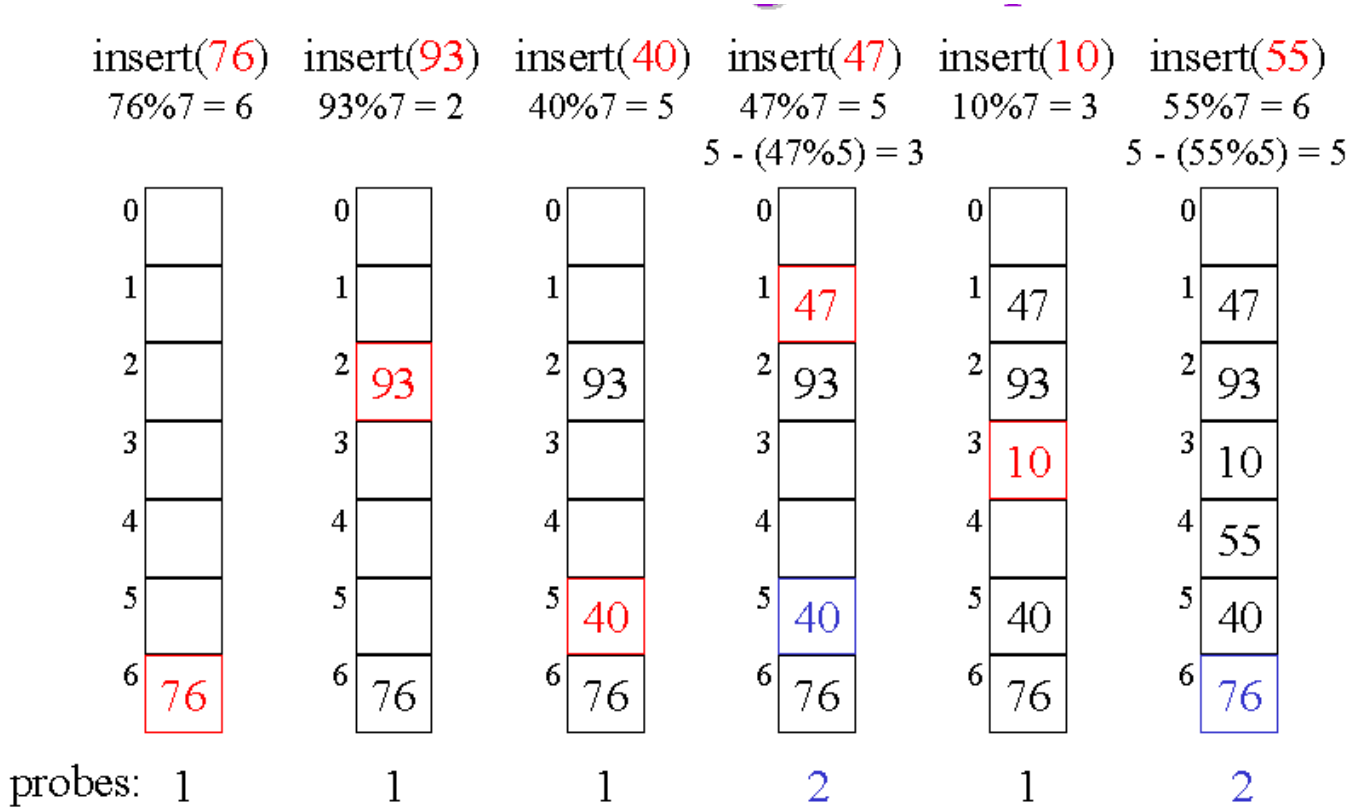
Quadratic Probing:

- Quadratic probing is another open addressing technique that probes buckets in a quadratic sequence when a collision occurs.
- It checks buckets at positions with increasing quadratic intervals (e.g., +1, +4, +9, +16, ...) until an empty bucket is found.



Double Hashing:

- Double hashing is a more complex open addressing technique. It uses a second hash function to determine the step size for probing.
- When a collision occurs, the algorithm applies the second hash function to the key and uses its output as the step size to find the next open bucket.



Each collision resolution method has its advantages and disadvantages, and the choice of the appropriate method depends on factors like the expected number of collisions, load factor, memory overhead, and performance requirements. An effective collision resolution strategy is essential to ensure efficient data retrieval and storage in hash-based data structures like hash tables.

Inserting and Deleting element from Hash table

Inserting and deleting elements from a hash table data structure involves using the hash function to determine the index where the element should be stored or removed. The process varies slightly depending on the collision resolution technique used. Let's explore how to insert and delete elements in a hash table using separate chaining and open addressing collision resolution methods:

Insertion in Hash Table:

a. Separate Chaining:

- Compute the hash code for the given key using the hash function.
- Convert the hash code to a valid index within the array using the hash code to index conversion (commonly modulo operation with the array size).
- Traverse the linked list (if it exists) at the computed index to check if the key already exists in the hash table.

- If the key exists, update the associated value with the new value (optional, depending on the application).
- If the key does not exist, create a new node with the key-value pair and insert it at the beginning or end of the linked list at the computed index.

b. Open Addressing:

- Compute the hash code for the given key using the hash function.
- Convert the hash code to a valid index within the array using the hash code to index conversion.
- If the computed index is empty (i.e., no collision), insert the key-value pair directly into the bucket at the computed index.
- If the computed index is occupied due to a collision, use the chosen open addressing technique (e.g., linear probing, quadratic probing, double hashing) to find the next available bucket to insert the key-value pair.

Deletion from Hash Table:

a. Separate Chaining:

- Compute the hash code for the given key using the hash function.
- Convert the hash code to a valid index within the array using the hash code to index conversion.
- Traverse the linked list (if it exists) at the computed index to find the node containing the key to be deleted.
- If the key is found, remove the node from the linked list, and update the pointers accordingly.

b. Open Addressing:

- Compute the hash code for the given key using the hash function.
- Convert the hash code to a valid index within the array using the hash code to index conversion.
- Use the open addressing technique to search for the key starting from the computed index.
- If the key is found, mark the bucket as "deleted" (a special flag indicating a deleted element) or remove the key-value pair from the bucket.
- Continue searching until the key is found or an empty bucket is encountered.

It's important to note that, in both insertion and deletion, the hash function must consistently produce the same hash code for a given key. Additionally, when using open addressing, care must be taken to handle the specific conditions of the chosen open addressing method, such as quadratic probing or double hashing, to ensure proper element retrieval and removal.

Properly implementing the insertion and deletion operations is crucial to maintaining the integrity and efficiency of the hash table data structure.