HASHING, SETS, AND MAPS

Hash Functions and Table Design

Hashing is the process of converting a given key into another value using a hash function.

Hash Table

or a **hash map**, maps **keys** to **values** using a **hash function**. It achieves fast operations (insertion, search, and deletion) by calculating an index for each key, ensuring efficient storage and retrieval of the value.

To visualize:

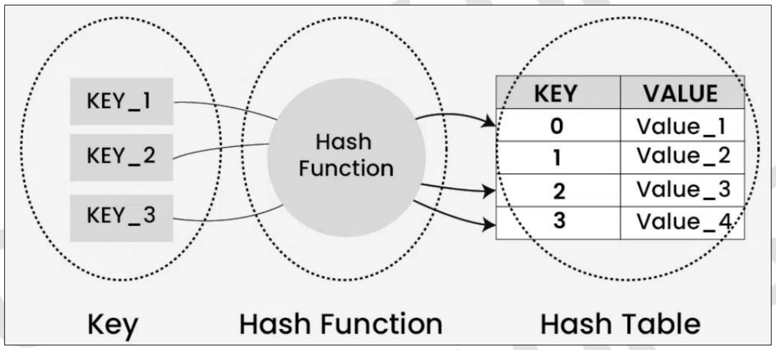

Figure 1. Hash table. Retrieved from https://www.geeksforgeeks.org/dsa/hash-table-data-structure/

A hash table is preferred over arrays or linked lists because operations are done much more quickly, even for large amounts of data. In a linked list, finding a person `"Jack"` takes more time as it would need to go from a node to the next and check until the node with `"Jack"` is found. With an array, it can be fast, but only if the index is known. But if the name `"Jack"` is known, it must be compared with each element, which takes time. But with a hashing table, finding `"Jack"` is done quicker because there is a way to go directly to where `"Jack"` is stored using the hash function.

Real-life Analogy:
Imagine a library with numbered shelves:

- Instead of walking through the whole library to find a book,
- You have a magical machine (hash function) that instantly tells you which shelf number to check.
- You go directly to that shelf, grab the book (value), and are done.

This makes finding things almost instant.

Hash tables work based on the following:

1. **Key-Value Pair**
   Data is stored as pairs of keys and values, such as:

   Key: `"name"`, Value: `"Kimmy"`
   Key: `"age"`, Value: `32`
   Key: `"city"`, Value: `"Tacloban"`

2. **Hash Function**
   A hash function generates an integer (hash value) from the key. Additionally, it is a mathematical function that takes an input (or "key") of arbitrary size and produces a fixed-size numerical output, called a **hash**, **hash code,** or **hash value**.

`hash("name")` -> 2 => 2 is the hash value
`hash("age")` -> 4 => 4 is the hash value
`hash("city")` -> 3 => 3 is the hash value

To ensure the hash value fits within the hash table array size, the formula is :

```
index = hash(key) % array_size or h(k) = k mod m
wherein, h(k) = hash value, k = key, mod = %, and m = number of buckets
```

For example:

```
k = 2
m = 10
h(27)=27 mod 10=7
```

Meaning: store key 27 in bucket 7.

3. **Storage in Buckets**
   The index determines the **bucket** where the key-value pair will be stored. Buckets are simply array slots.

Index	Bucket
0	
1	
2	`("name", "Kimmy")`
3	`("city", "Tacloban")`
4	`("age", 32)`

4. **Retrieval**
   The same hash function is used to locate the bucket using the key, and the value is retrieved from that bucket.

This is a sample Java program using `HashMap`, a hash table-based implementation of the `Map` interface.

```
import java.util.HashMap;

public class HashTableExample {

    public static void main(String[] args) {

        // 1. Create a HashMap (Java's hash table)
        HashMap<String, Integer> hashTable = new HashMap<>();

        // 2. Insert key-value pairs
        hashTable.put("apple", 10);
        hashTable.put("banana", 20);
        hashTable.put("grape", 30);

        // 3. Retrieve values using keys
        System.out.println("apple: " + hashTable.get("apple"));
        System.out.println("banana: " + hashTable.get("banana"));

        // 4. Check if a key exists
        if (hashTable.containsKey("cherry")) {
            System.out.println("cherry: " + hashTable.get("cherry"));
        } else {
            System.out.println("cherry not found!");
        }
        // 5. Remove a key-value pair
        hashTable.remove("grape");

        // 6. Display entire hash table
        System.out.println("Hash table contents: " + hashTable);
    }
}
```

Output:

```
apple: 10
banana: 20
cherry not found!
Hash table contents: {apple=10, banana=20}
```

Here are the operations used:

- Insertion: `put(key, value)` – adds a new key-value pair to the hash table.
- Retrieval: `get(key)` – returns the value associated with the key.
- Checking: `containsKey` – checks if the given key exists in the hash table.
- Deletion: `remove(key)` – removes the key-value pair associated with the specified key.

Table Design

The term **table design** refers to how this internal table is structured, sized, and organized so that:

- Keys are distributed as evenly as possible.
- Collisions (when two keys map to the same bucket) are minimized and efficiently resolved.
- Memory is used efficiently while maintaining average-case performance for lookup, insertion, and deletion.

Components of Table Design

Table Size Selection

- **Initial Size:** Choose the number of buckets (m) carefully.
    - If `m` is too small, it will result in too many collisions, which degrades the performance.
    - If `m` is too large, it will result in memory waste.
- **Prime Numbers:** When using division-based hash functions (`h(k) = k mod m`), selecting `m` as a prime number reduces clustering.
- **Resizing:** Many implementations **dynamically resize** (rehash) when the **load factor** (ratio of stored entries to table size) exceeds a threshold (e.g., 0.75).

Hash Function Selection

The hash function must:

- Be deterministic, as the same key must have the same hash value.
- Distribute keys uniformly to minimize clustering.
- Be fast to compute to keep insertions/lookups efficient.

Bucket Structure

Each position in the table is called a **bucket**. Table design specifies how each bucket stores data:

1. **Chaining:**
    - Each bucket contains a linked list (or tree) of key-value pairs.
    - Multiple keys that hash to the same index are stored in the list.

- Lookup time increases slightly with the length of the list.
- Easy to implement and handles a high number of collisions gracefully.

2. **Open Addressing:**
   - Each bucket holds only one (1) entry.
   - If a collision occurs, probing is used to find another empty bucket:
     - **Linear Probing:** Check the next slot sequentially.
     - **Quadratic Probing:** Check slots using quadratic steps.
     - **Double Hashing:** Use a second hash function to compute probe steps.
   - Requires careful handling of deletions (to avoid breaking probe chains).

Load Factor Management
The **load factor** measures how "full" a hash table is. It is computed by getting the quotient between the number of stored elements over the number of buckets. It monitors and controls the ratio to maintain good performance.

If a **low load factor** occurs, the table is mostly empty, and only a few collisions. The table needs to **grow**. If a **high load factor** occurs, the table is crowded and collisions become more likely, which degrades the performance. The table needs to be **rehashed**, which creates a larger table and re-computes indices for all keys.

Collision Handling Techniques

Collisions happen when two (2) different keys are processed by the hash function and produce the same index (bucket) in the table. Since each bucket typically holds only one index position, the hash table must handle this situation so that both keys and their values can still be stored and retrieved correctly.

To visualize:

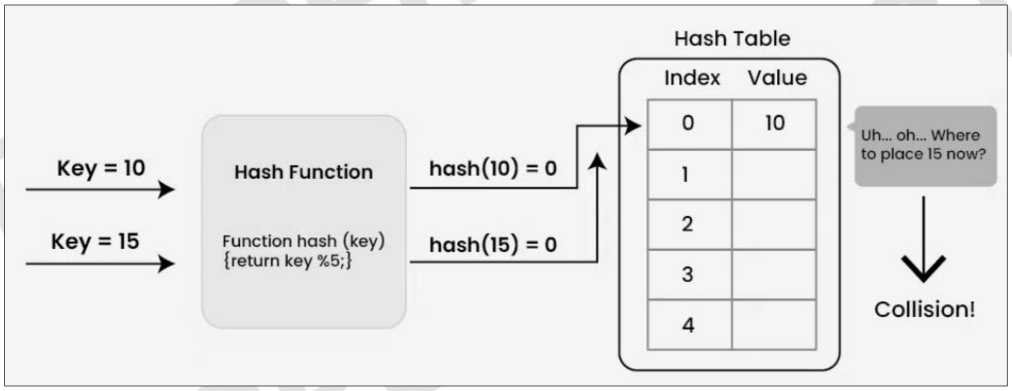

Figure 2. Collision. Retrieved from https://www.geeksforgeeks.org/dsa/hash-table-data-structure/

Collision Handling Techniques

Chaining
This stores multiple key-value pairs in the same bucket using a secondary data structure, most commonly a linked list, but sometimes a balanced tree (e.g., Java HashMap uses trees when buckets get too long). The idea is to make each cell of the hash table point to a linked list of records that have the same hash function values.

Process:

1. Compute the hash using: `h(k) = k mod m`
2. Go to bucket `h(k)`.
   - If the bucket is empty, create a new list and insert the key-value pair.
   - If the bucket is occupied, append the new key-value pair to the list.
3. Search by computing the `h(k)` and searching for the matching key within the list of that bucket.

For example:

Assume 5 buckets, insert 12, 27, 32 with `h(k)=k mod 5`

Key	h(k)	Bucket
12	2	2
27	2	2
32	2	2

Bucket 2 now contains: [(12, A), (27, B), (32, C)]

Open Addressing
In this technique, the values are all stored in the hash table itself. When a collision occurs, the algorithm searches for the next available slot using a **probing sequence**. The table size should be greater than the number of keys at all times. It is used when there are space restrictions.

Types of Open Addressing:

- **Linear Probing**: If the desired bucket is occupied, check the **next slot sequentially** until an empty one is found.
- **Quadratic Probing**: Instead of moving one slot at a time, move in **quadratically increasing steps** to reduce clustering.
- **Double Hashing**: Uses a **second hash function** to compute the probe step size, producing a more random probe sequence.

Rehashing
Regardless of the collision method, hash tables often use rehashing when the load factor exceeds a threshold:

- Create a new table (usually double the size).
- Recompute the hash for each key.
- Insert all keys into the new table.

Set Operations and Implementations

A **Set Data Structure** is a data structure that stores a collection of unique elements. It is used to store and manipulate a group of objects, where each object is **unique**. It does not allow duplicates; if done so, the set ignores it.

Types of Set Data Structure
Unordered Set
An unordered associative container implemented using a hash table where keys are hashed into indices of a hash table, so that the insertion is always randomized. All operations on the unordered set take constant time on average, which can go up to linear time in the worst case. This depends on the internally used hash function, but practically, they perform well and provide a constant-time lookup operation.

Ordered Set
The most common set data structure. It is generally implemented using balanced BSTs, and it supports lookups, insertions, and deletion operations.

A set can be implemented in various ways:

1. **Hash-based Set:** The set is represented as a hash table where each element in the set is stored in a bucket based on its hash code.

   Sample Java Program:

```
import java.util.HashSet;

public class HashSetExample {
    public static void main(String[] args) {
        HashSet<String> fruits = new HashSet<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Apple");         // Duplicate, ignored
        System.out.println(fruits);
    }
}
```

   Output:

```
[Banana, Apple]
```

   or

```
[Apple, Banana]
```

   - A HashSet does not guarantee order. Elements are stored based on hash codes, not insertion order.
   - The output will contain both **"Apple"** and **"Banana"**, but their order may vary depending on the hash table bucket assignment.

2. **Tree-based Set:** The set is represented as a binary search tree where each node in the tree represents an element in the set.

   Sample Java Program

```
import java.util.TreeSet;

public class TreeSetExample {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(50);
        numbers.add(10);
        numbers.add(30);
        System.out.println(numbers);
    }
}
```

   Output:

```
[10, 30, 50]
```

Unlike HashSet, TreeSet maintains elements in ascending order by default because it uses a binary search tree structure. This guarantees a predictable, sorted order every time you print or iterate over the set.

Set data structures are commonly used in algorithms, data analysis, and databases. The main advantage of using a set data structure is that it allows operations on a collection of elements in an efficient and organized way.

Map Data Structures

A **Map Data Structure** is a fundamental data structure that stores information as key–value pairs, where each key is unique and maps to exactly one value**.**

Maps, unlike arrays or lists, use unique keys to identify and access their associated values. This enables fast data retrieval and modification without the need to know the specific index or position. This is ideal for situations where fast lookups, insertions, and updates are needed.

Key Features of Map Data Structure

- **Key-Value Pairing:** Each element in the map consists of a unique key and a corresponding value.
- **Fast Access:** Maps provide efficient lookup and retrieval operations based on the key. They can achieve constant or logarithmic time complexity for common operations.
- **Dynamic Size:** Maps can dynamically grow and shrink in size as elements are added or removed. This flexibility makes them suitable for handling varying amounts of data.
- **Uniqueness of Keys:** Each key within a map must be unique to ensure that every key-value pair represents a distinct entry.

Types of Maps in Data Structures

Hash Map
A data structure that maps keys to array indices using a hash function to convert each key into a numeric index, allowing values to be stored and retrieved in constant time.

Java Syntax:

```
Map<KeyType, ValueType> map = new HashMap<>();
```

Java Program Example:

```
import java.util.Map;       // Import the Map interface
import java.util.HashMap;   // Import the HashMap class

public class HashMapExample {
    public static void main(String[] args) {
        // Create a HashMap object using the Map interface
        // Key type: String, Value type: String
        Map<String, String> map = new HashMap<>();

        // ----- INSERTION -----
        // put(key, value) inserts a key-value pair into the map.
        // If the key already exists, the value will be updated.
        map.put("name", "Kimmy");
        map.put("course", "BSIT");
        // Print the entire map.
        // Output format: {key1=value1, key2=value2}
        System.out.println("HashMap contents: " + map);

        // ----- RETRIEVAL -----
        // get(key) retrieves the value associated with a given key.
        String studentName = map.get("name");
        System.out.println("Retrieved value for 'name': " + studentName);

        // ----- MEMBERSHIP CHECK -----
        if (map.containsKey("course")) {
            System.out.println("'course' key exists with value: " + map.get("course"));
        }

        // ----- DELETION -----
        map.remove("course");
        System.out.println("HashMap after removing 'course': " + map);
    }
}
```

Output:

```
HashMap contents: {course=BSIT, name=Kimmy}
Retrieved value for 'name': Kimmy
'course' key exists with value: BSIT
HashMap after removing 'course': {name=Kimmy}
```

Tree Map
A map that uses a binary search tree as its implementation. The keys in a tree map are stored in sorted order, which allows for efficient searching, insertion, and deletion operations.

Java Syntax:

```
Map<KeyType, ValueType> map = new TreeMap<>();
```

Java Program Example:

```
import java.util.Map;
import java.util.TreeMap;

public class TreeMapExample {
    public static void main(String[] args) {
        // Create a TreeMap with String keys and Integer values
        Map<String, Integer> treeMap = new TreeMap<>();

        // Insert key-value pairs (automatically stored in sorted order)
        treeMap.put("banana", 20);
        treeMap.put("apple", 10);
        treeMap.put("grape", 30);

        // Display the TreeMap
        System.out.println("TreeMap contents: " + treeMap);
    }
}
```

Output:

```
TreeMap contents: {apple=10, banana=20, grape=30}
```

Linked Hash Map

A linked hash map combines hashing with a linked list to preserve insertion order, making iteration predictable. This enables rapid iteration over the map's elements, as well as efficient insertion, retrieval, and deletion operations.

Java Syntax:

```
Map<KeyType, ValueType> map = new LinkedHashMap<>();
```

Java Program Example:

```
import java.util.Map;
import java.util.LinkedHashMap;

public class LinkedHashMapExample {
    public static void main(String[] args) {
        // Create a LinkedHashMap with String keys and Integer values
        Map<String, Integer> linkedMap = new LinkedHashMap<>();

        // Insert key-value pairs (order is preserved)
        linkedMap.put("banana", 20);
        linkedMap.put("apple", 10);
        linkedMap.put("grape", 30);

        // Display the LinkedHashMap
        System.out.println("LinkedHashMap contents: " + linkedMap);
    }
}
```

Output:

```
LinkedHashMap contents: {banana=20, apple=10, grape=30}
```

References:

De Silva, K. (2024). *Data structures and algorithms: Hash table*. Retrieved on September 3, 2025, from https://medium.com/@kalanamalshan98/data-structures-and-algorithms-hash-table-94ca7f21a341

Oracle. (2025). *Hashtable.* Retrieved on September 3, 2025, from https://docs.oracle.com/javase/8/docs/api/java/util/Hashtable.html

Prepbytes. (2023). *What is a map data structure?* Retrieved on September 4, 2025, from https://prepbytes.com/blog/what-is-map-data-structure/