HARAMAYA UNIVERSITY

COLLEGE OF COMPUTING AND INFORMATICS

DEPARTMENT OF SOFTWARE ENGINEERING

ASSIGNMENT FOR DESIGN AND ANALYSIS OF ALGORITHM

NAME	ID NUMBER

1. **Ruot chuol 5037/14**
2. **Hawo Mohammed 5067/14**
3. **Yusuf kedir 3737/14**

1. **Hashing (Overview, pros & cons, Hashing Methods, hash tables, hashing functions)**

➢ Introduction to hashing

Hashing is a technique used in data structures that efficiently stores and retrieves data in a way that allows for quick access. It involves mapping data to a specific index in a hash table using a hash function that enables fast retrieval of information based on its key. This method is commonly used in databases, caching systems, and various programming applications to optimize search and retrieval operations. The great thing about hashing is, we can achieve all three operations (search, insert and delete) in O(1) time on average.

A hash data structure is a type of data structure that allows for efficient insertion, deletion, and retrieval of elements. It is often used to implement associative arrays or mappings, which are data structures that allow you to store a collection of key-value pairs.

In a hash data structure, elements are stored in an array, and each element is associated with a unique key. To store an element in a hash, a hash function is applied to the key to generate an index into the array where the element will be

stored. The hash function should be designed such that it distributes the elements evenly across the array, minimizing collisions where multiple elements are assigned to the same index.

Example. H(X), is hush function

List = [11,12,13,14,15], H(X) = [X%10]

H(11) = 11%10 = 1, H(12) = 12%10 = 2, H(13) = 13%10 = 3

H(14) = 14%10 = 4, H(15) = 15%10 = 5

Hash Table

0	1	2	3	4	5
10	11	12	13	14	15

When retrieving an element from a hash, the hash function is again applied to the key to find the index where the element is stored. If there are no collisions, the element can be retrieved in constant time, O(1). However, if there are collisions, multiple elements may be assigned to the same index, and a search must be performed to find the correct element.

To handle collisions, there are several strategies that can be used, such as chaining, where each index in the array stores a linked list of elements that have collided, or open addressing, where collisions are resolved by searching for the next available index in the array.

Hash data structures have many applications in computer science, including implementing symbol tables, caches, and databases. They are especially useful in situations where fast retrieval of elements is important, and where the number of elements to be stored may be large.

Collision resolution:

Collisions happen when two or more keys point to the same array index. Chaining, open addressing, and double hashing are a few techniques for resolving collisions.

collision resolution in hash can be done in two methods:

- Open Addressing , and
- Closed Addressing

1. **Open Addressing:** Open addressing collision resolution technique involves generating a location for storing or searching the data called probe. It can be done in the following ways:

✓ **Linear Probing:** If there is a collision at **i** then we use the hash function – **H(k, i) = [H'(k) + i] % m**
where, **i** is the index, **m** is the size of hash table **H(k, i)** and **H'(k)** are hash functions.

✓ **Quadratic Probing:** If there is a collision at **i** then we use the hash function – **$H(k, i) = [H'(k) + c_1 * i + c_2 * i^2] \% m$**
where, **i** is the index, **m** is the size of hash table **H(k, i)** and **H'(k)** are hash functions, **c_1** and **c_2** are constants.

✓ **Double Hashing:** If there is a collision at **i** then we use the hash function – **$H(k, i) = [H_1(k, i) + i * H_2(k)] \% m$**
where, **i** is the index, **m** is the size of hash table **H(k, i)**, **$H_1(k) = k \%$ m** and **$H_2(k) = k \% m'$** are hash functions

2. **Closed Addressing:**

Closed addressing collision resolution technique involves chaining. Chaining in the hashing involves both array and linked list. In this method, we generate a probe with the help of the hash function and link the keys to the respective index one after the other in the same index. Hence, resolving the collision.

Need for Hash data structure

Every day, the data on the internet is increasing multifold and it is always a struggle to store this data efficiently. In day-to-day programming, this amount of data might not be that big, but still, it needs to be stored, accessed, and processed easily and efficiently. A very common data structure that is used for such a purpose is the Array data structure.

Now the question arises if Array was already there, what was the need for a new data structure! The answer to this is in the word " efficiency ". Though storing in Array takes O(1) time, searching in it takes at least O(log n) time. This time appears to be small, but for a large data set, it can cause a lot of problems and this, in turn, makes the Array data structure inefficient.

So now we are looking for a data structure that can store the data and search in it in constant time, i.e. in O(1) time. This is how Hashing data structure came into play. With the introduction of the Hash data structure, it is now possible to easily store data in constant time and retrieve them in constant time as well

Components of hashing

There are majorly three components of hashing

1. *Key:* A key can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.
2. *Hash Function:* the hush function receives the input key and returns the index of an element in an array called a hash table. The index is known as the hash index.
3. *Hash Table:* Hash table is a data structure that maps keys to values using a special function called a hash function. Hash stores the data in an associative manner in an array where each data value has its own index.

➢ Hash Functions and Types of Hash functions.

Hash functions are a fundamental concept in computer science and play a crucial role in various applications such as data storage, retrieval, and cryptography. In data structures and algorithms (DSA), hash functions are primarily used in hash tables, which are essential for efficient data management. This article delves into the intricacies of hash functions, their properties, and the different types of hash functions used in DSA.

A hash function is a function that takes an input (or 'message') and returns a fixed-size string of bytes. The output, typically a number, is called the hash code or hash value. The main purpose of a hash function is to efficiently map data of arbitrary size to fixed-size values, which are often used as indexes in hash tables.

Key Properties of Hash Functions

Deterministic: A hash function must consistently produce the same output for the same input.

Fixed Output Size: The output of a hash function should have a fixed size, regardless of the size of the input.

Efficiency: The hash function should be able to process input quickly.

Uniformity: The hash function should distribute the hash values uniformly across the output space to avoid clustering.

Pre-image Resistance: It should be computationally infeasible to reverse the hash function, i.e., to find the original input given a hash value.

Collision Resistance: It should be difficult to find two different inputs that produce the same hash value.

Avalanche Effect: A small change in the input should produce a significantly different hash value.

Applications of Hash Functions

Hash Tables: The most common use of hash functions in DSA is in hash tables, which provide an efficient way to store and retrieve data.

Data Integrity: Hash functions are used to ensure the integrity of data by generating checksums.

Cryptography: In cryptographic applications, hash functions are used to create secure hash algorithms like SHA-256.

Data Structures: Hash functions are utilized in various data structures such as Bloom filters and hash sets.

Types of Hash Functions

There are many hash functions that use numeric or alphanumeric keys. This article focuses on discussing different hash functions:

1. Division Method.
2. Multiplication Method
3. Mid-Square Method
4. Folding Method
5. Cryptographic Hash Functions
6. Universal Hashing
7. Perfect Hashing

Let's begin discussing these methods in detail.

1. Division Method

The division method involves dividing the key by a prime number and using the remainder as the hash value.

h(k) = k mod m

Where k is the key and m is a prime number.

Advantages:

Simple to implement.

Works well when mm is a prime number.

Disadvantages:

Poor distribution if mm is not chosen wisely.

2. Multiplication method

In the multiplication method, a constant A (0 < A < 1) is used to multiply the key. The fractional part of the product is then multiplied by m to get the hash value.

h(k)=⌊m(kAmod1)⌋

Where ⌊ ⌋ denotes the floor function.

Advantages: Less sensitive to the choice of mm.

Disadvantages: More complex than the division method.

3. Mid-Square Method

In the mid-square method, the key is squared, and the middle digits of the result are taken as the hash value.

Steps:

1. Square the key.
2. Extract the middle digits of the squared value.

Advantages: Produces a good distribution of hash values.

Disadvantages: May require more computational effort.

4. Folding Method

The folding method involves dividing the key into equal parts, summing the parts, and then taking the modulo with respect to m.

Steps:

1. Divide the key into parts.
2. Sum the parts.
3. Take the modulo *m* of the sum.

Advantages: Simple and easy to implement.

Disadvantages: Depends on the choice of partitioning scheme.

5. Cryptographic Hash Functions

Cryptographic hash functions are designed to be secure and are used in cryptography. Examples include MD5, SHA-1, and SHA-256.

Characteristics:

Pre-image resistance.

Second pre-image resistance.

Collision resistance.

Advantages: High security.

Disadvantages: Computationally intensive.

6. Universal Hashing

Universal hashing uses a family of hash functions to minimize the chance of collision for any given set of inputs.

$h(k)=((a\cdot k+b) \bmod p) \bmod m$

Where a and b are randomly chosen constants, p is a prime number greater than m, and k is the key.

Advantages: Reduces the probability of collisions.

Disadvantages: Requires more computation and storage.

7. Perfect Hashing

Perfect hashing aims to create a collision-free hash function for a static set of keys. It guarantees that no two keys will hash to the same value.

Types:

- ✓ Minimal Perfect Hashing: Ensures that the range of the hash function is equal to the number of keys.
- ✓ Non-minimal Perfect Hashing: The range may be larger than the number of keys.

Advantages: No collisions.

Disadvantages: Complex to construct.

In conclusion, hash functions are very important tools that help store and find data quickly. Knowing the different types of hash functions and how to use them correctly is key to making software work better and more securely. By choosing the right hash function for the job, developers can greatly improve the efficiency and reliability of their systems.

➢ Hash Table

A Hash table is defined as a data structure used to insert, look up, and remove key-value pairs quickly. It operates on the hashing concept, where each key is translated by a hash function into a distinct index in an array. The index functions as a storage location for the matching value. In simple words, it maps the keys with the value.

Hash tables are a versatile and efficient data structure widely used in computer science. By understanding their mechanics—hash functions, collision handling, and performance characteristics—developers can leverage them effectively for various applications.

➢ Hashing Methods

Hashing methods are techniques used to map data of varying sizes into fixed-size values, typically for efficient data retrieval in hash tables. Here's a discussion of several common hashing methods:

1. **Division Method:** this method uses the modulo operator to compute the hash code.

Formula: hash(k) = k mod m, where k is the key and m is the size of the hash table.

Characteristics:

- ✓ it is simple and easy to implement.
- ✓ The choice of m (should be a prime number) affects the distribution of hash codes and can help reduce collisions.

2. **Multiplication Method:** This method multiplies the key by a constant and then takes the fractional part.
   Formula: $hash(k)=\lfloor m\cdot(k\cdot A \bmod 1)\rfloor$, where A is a constant ($0 < A < 1$) and m is the size of the hash table.
   Characteristics:

✓ Provides a good distribution of hash values
✓ Less sensitive to the choice of m compared to the division method.

3. **MurmurHash Method:** A non-cryptographic hash function known for its speed and good distribution.
   Characteristics:
   - ✓ Often used in hash-based data structures and databases.
   - ✓ Works well with arbitrary data types and sizes.
   - ✓ Generates a hash value that minimizes collisions.

4. **Cryptographic hash functions:** designed for security perposes, these functions generate a fixed size hash value from input data. Example, SHA-256, MD5.
   Characteristics:
   - ✓ Resistant to collisions (two different inputs should not produce the same output).
   - ✓ Computationally intensive, making them slower than non-cryptographic hashes.
   - ✓ Useful in applications requiring data integrity, such as digital signatures and password hashing.

5. **Open Addressing :** a collision resolution method, where upon a collision, the algorithm searches for the next available slot in the hash table.
   Techniques:
   - ✓ Linear probing: Check the next slot sequentially.

- ✓ Quadratic Probing: Check slots based on a quadratic function, reducing clustering issues.
- ✓ Double Hashing: use a second hash function to determine the step size for probing.

6. **Chaining :** A collision resolution method where each slot in the hash table contains a linked list (or another data structure) of entries that hash to the same index.

Characteristics:

- ✓ Allows multiple elements to be stored at the same index.
- ✓ Simple to implement and can handle a large number of collisions effectively.
- ✓ Performance depends on the load factor (number of entries / number of slots).

7. **Perfect Hashing:** A method that aims to create a hash function that maps keys to unique indices without collisions.

Characteristics:

- ✓ Often used when the set of keys is known in advance.
- ✓ Can be implemented using a secondary hash function to resolve collisions for a specific set of keys.
- ✓ Provides O(1) time complexity for search operations.

In conclusion, Hashing methods vary in complexity, performance, and use cases. Choosing the right hashing method involves considering factors such as the expected number of elements, the need for security, and the nature of the data being handled. Each method has its strengths and weaknesses, making them suitable for different applications in data structures and algorithms.

➢ Pros and cons of hashing

❖ Pros of hashing

1. *Fast access time:* Average cost of O(1), hash tables allow for average case constant time complexity for search, insert, and delete operations, making them ideal for applications requiring quick data retrieval.

2. *Efficient memory usage:* Hashing can lead to better memory utilization compared to some other data structures, as it allows for dynamically sized arrays.

3. *Scalability:* Hash tables can grow dynamically to accommodate more data, making them suitable for applications with fluctuating data sizes.

4. *Key-based access:* Hashing allows for direct access to data via unique keys, simplifiying data retrieval and management.

5. *Collision resolution:* techniques such as chaining and open addressing provide flexibility in handling collision, allowing the data structure to maintain performance even when multiple keys hash to the same index.

6. *Support for complex data types:* hashing can be applied to various data types,including strings and objects, making it versatile in different programming contexts.

❖ **Cons of Hashing**

1. *Collisions:* Despite collision resolution techniques, collisions can still occur, which may degrade performance and lead to higher time complexities in worst case.

2. *Space overhead:* hash tables may require extra space for the structure(like linked lists for chaining) and can waste memory if the load factor is not managed effectively.

3. *Complexity of implementation:* Implementing a hash table with efficient collision resolution and resizing strategies can be complex compared to simpler data structures like arrays or linked lists.

4. *Hash function dependancy:* the performance of a hash table heavily depends on the quality of the hash function. Poorly designed hash functions can lead to many collisions and uneven distribution, affecting efficiency.

5. *Difficulty in range queries:* Hash tables do not maitain any order among elements, making it difficult to perform range queries or ordered data retrieval.

6. *Rehashing overhead:* when the load factor exceeds a certain threshold, resizing(rehashing) the table can be an expensive operation, requiring all elements to rehashed and inserted into a new table.

7. *Not suitable for all applications:* for certain applications such as those that require ordered data,other data structures like trees(e.g.,binary search trees) can be more appropriate.