**303105201 - Design of Data Structures**

**UNIT-7 HASHING**

Hashing:

Hash Table organizations

Hashing Functions

Static and Dynamic Hashing

***What is Hashing?***

A hashing algorithm is used to convert an input (such as a string or integer) into a fixed-size output (referred to as a hash code or hash value). The data is then stored and retrieved using this hash value as an index in an array or hash table.

Hashing is commonly used to create a unique identifier for a piece of data, which can be used to quickly look up that data in a large dataset.

For example, a web browser may use hashing to store website passwords securely. When a user enters their password, the browser converts it into a hash value and compares it to the stored hash value to authenticate the user.

***What is a hash Key?***

In the context of hashing, a hash key (also known as a hash value or hash code) is a fixed-size numerical or alphanumeric representation generated by a hashing algorithm. It is derived from the input data, such as a text string or a file, through a process known as hashing.

***Hash Function:***

*Hash Function*: A hash function is a type of mathematical operation that takes an input (or key) and outputs a fixed-size result known as a hash code or hash value. The hash function must always yield the same hash code for the same input in order to be deterministic. Additionally, the hash function should produce a unique hash code for each input, which is known as the hash property.

unsigned int hashFunction(int key, int tableSize) {

return key % tableSize;

}

*Purpose:*

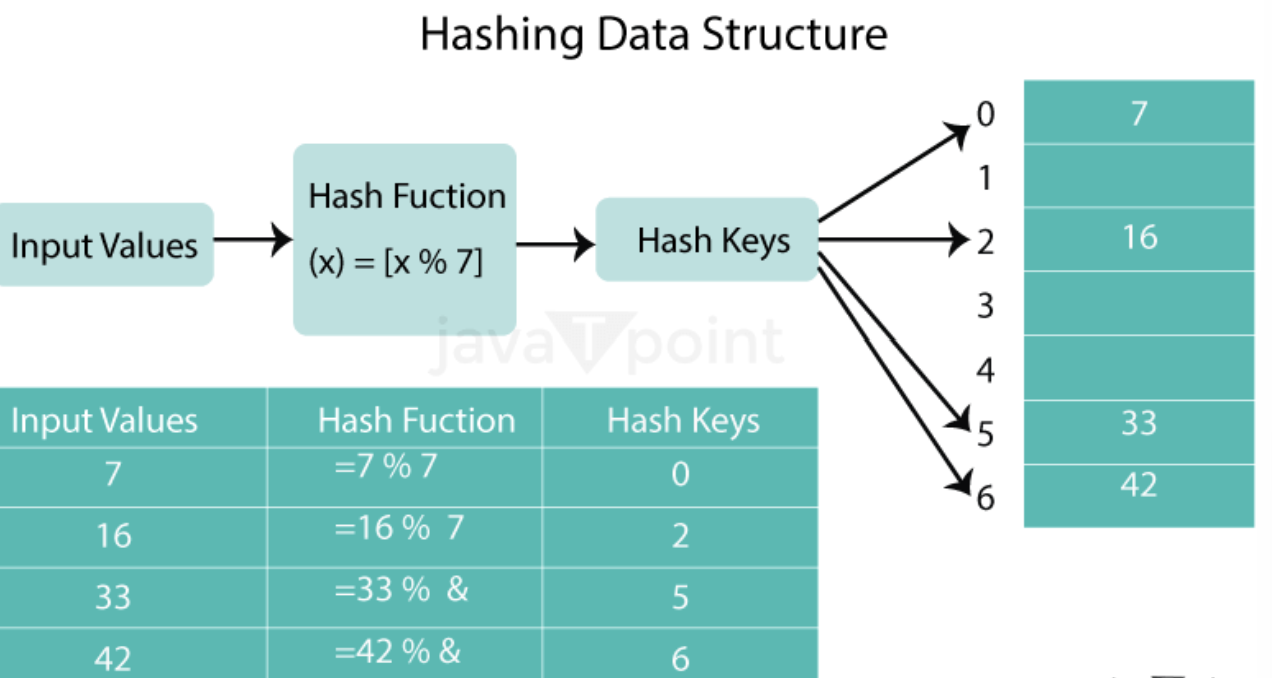
The purpose of the hash function is to map the key (which could be any integer) to a specific index within the bounds of the hash table (i.e., between 0 and tableSize - 1). This index is where the key (and associated value, if any) will be stored in the hash table.

return key % tableSize;

*% (Modulus Operator):*

The modulus operator returns the remainder of the division of the key by tableSize.

* Division method
* Multiplication method
* Universal hashing



*Division method:*

This method involves dividing the key by the table size and taking the remainder as the hash value. For example, if the table size is 10 and the key is 23, the hash value would be 3 (23 % 10 = 3).

H(K)=K mod M

K is the key value

M is the size of the hash table

* Hashing is an efficient method to store and retrieve elements.
* It’s exactly same as index page of a book.
* In index page, every topic is associated with a page number.
* If we want to look some topic, we can directly get the page number from the index

How to Calculate Hash key:-

Let’s take hash table size as 7

Size7

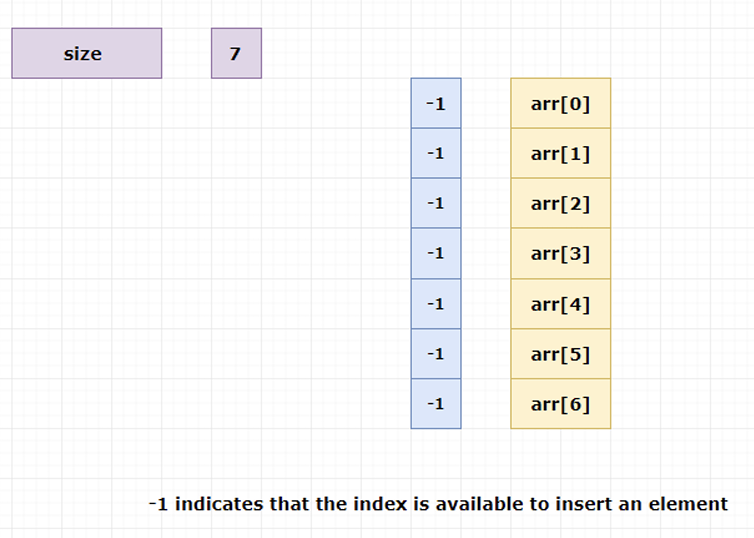
Arr[size];

Formula to calculate key is:

Key = element % size

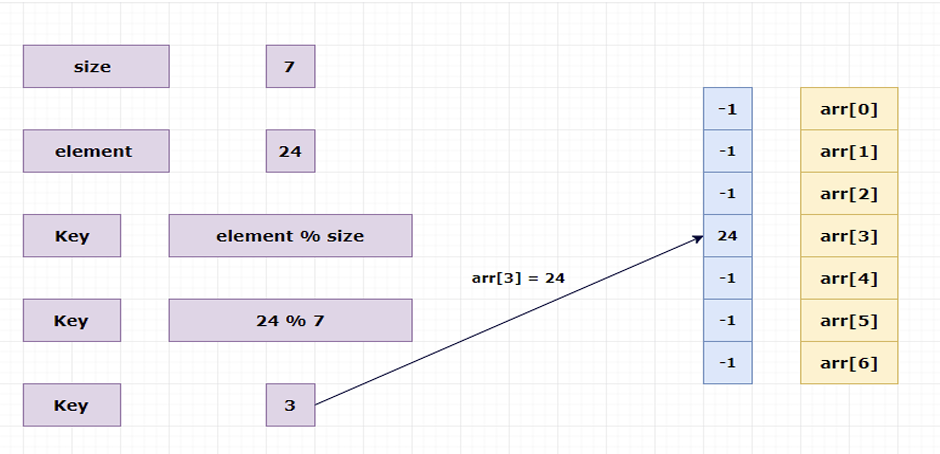
Initialize the Hash:

Before inserting elements into array. Let’s make array default value as -1. -1 indicates element not present or the particular index is available to insert.

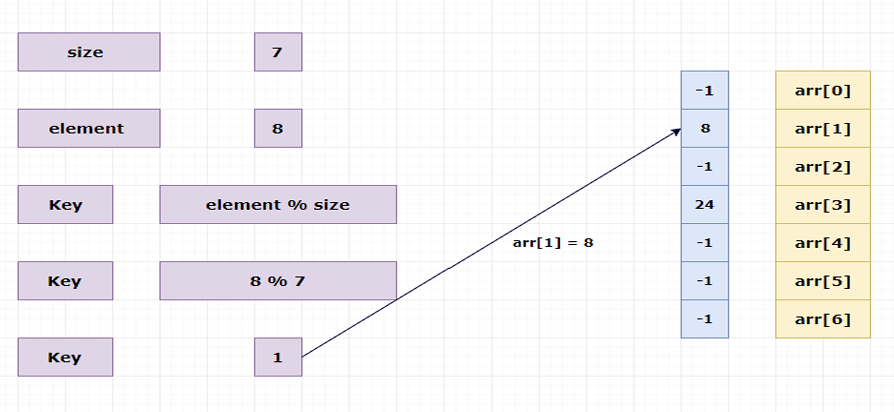


Inserting elements in the hash table

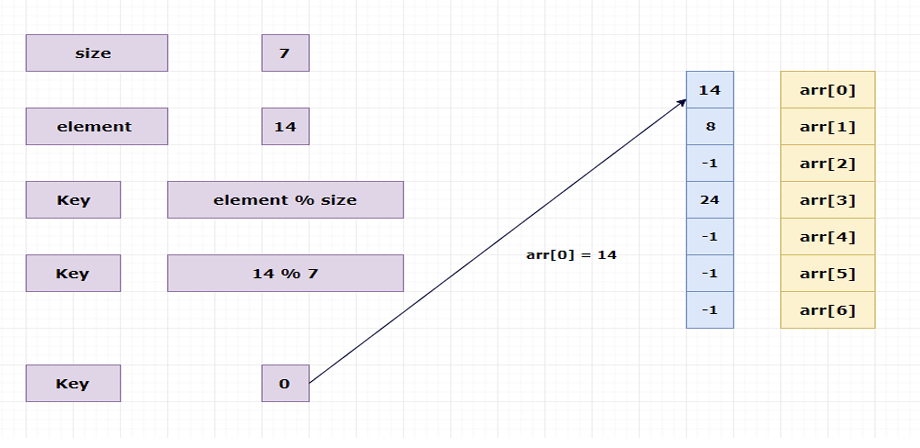
Insert 24



Insert 8

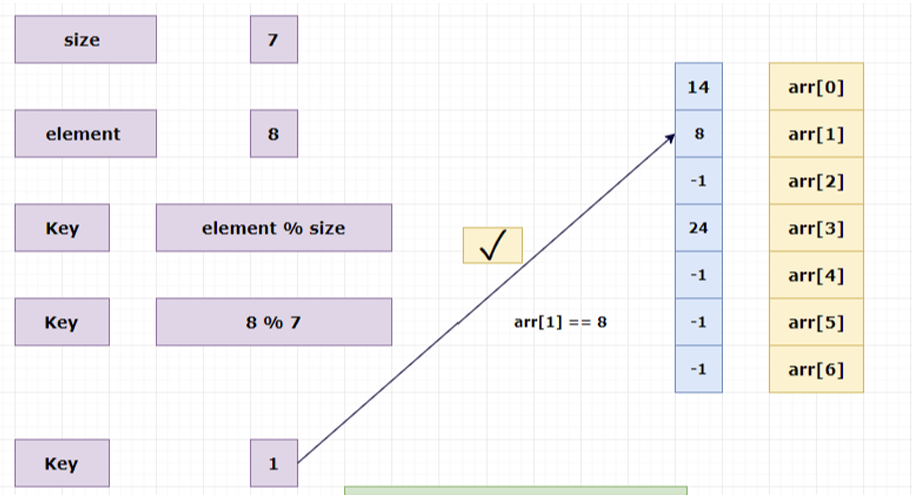


Insert 14



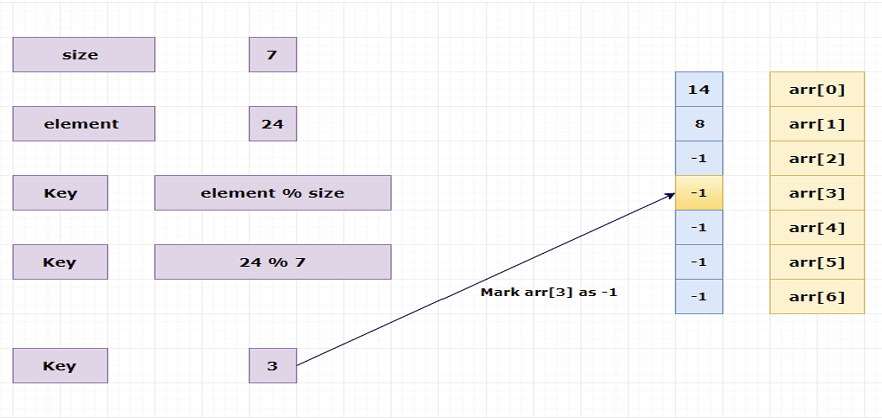
Searching elements from the hash table

Search 8



Deleting an element from the hash table

Delete 24



*Multiplication method:*

This method involves multiplying the key by a constant and taking the fractional part of the product as the hash value.

H(K)*=*floor(M(KA mod 1))

M is size of the hashing table

K is key value

A is constant

K=6, A=0.3, M=32

->KA

6\*0.3=1.8

->KA mod 1

1.8 mod 1=0.8

->M\*0.8

32\*0.8=25.6

->Floor(25.6)

=25

//h(6)=25

*Universal hashing:*

This method involves using a random hash function from a family of hash functions. This ensures that the hash function is not biased towards any particular input and is resistant to attacks.

***Hash Table Operations***

There are several operations that can be performed on a hash table, including:

*Insertion:* Inserting a new key-value pair into the hash table.

*Deletion:* Removing a key-value pair from the hash table.

*Search:* Searching for a key-value pair in the hash table.

***Creating a Hash Table***

Hashing is frequently used to build hash tables, which are data structures that enable quick data insertion, deletion, and retrieval. One or more key-value pairs can be stored in each of the arrays of buckets that make up a hash table.

To create a hash table, we first need to define a hash function that maps each key to a unique index in the array. A simple hash function might be to take the sum of the ASCII values of the characters in the key and use the remainder when divided by the size of the array. However, this hash function is inefficient and can lead to collisions (two keys that map to the same index).

***Applications of Hashing***

Hashing has many applications in computer science, including:

*Databases:* Hashing is used to index and search large databases efficiently.

*Password Storage:* Hash passwords before storing them, comparing hashes for authentication.

*Cryptography:* Hash functions are used to generate message digests, which are used to verify the integrity of data and protect against tampering.

*Caching:* Hash tables are used in caching systems to store frequently accessed data and improve performance.

*Spell checking:* Hashing is used in spell checkers to quickly search for words in a dictionary.

*Network routing:* Hashing is used in load balancing and routing algorithms to distribute network traffic across multiple servers.

***Advantages of Hashing:***

*Fast Access:* Hashing provides constant time access to data, making it faster than other data structures like linked lists and arrays.

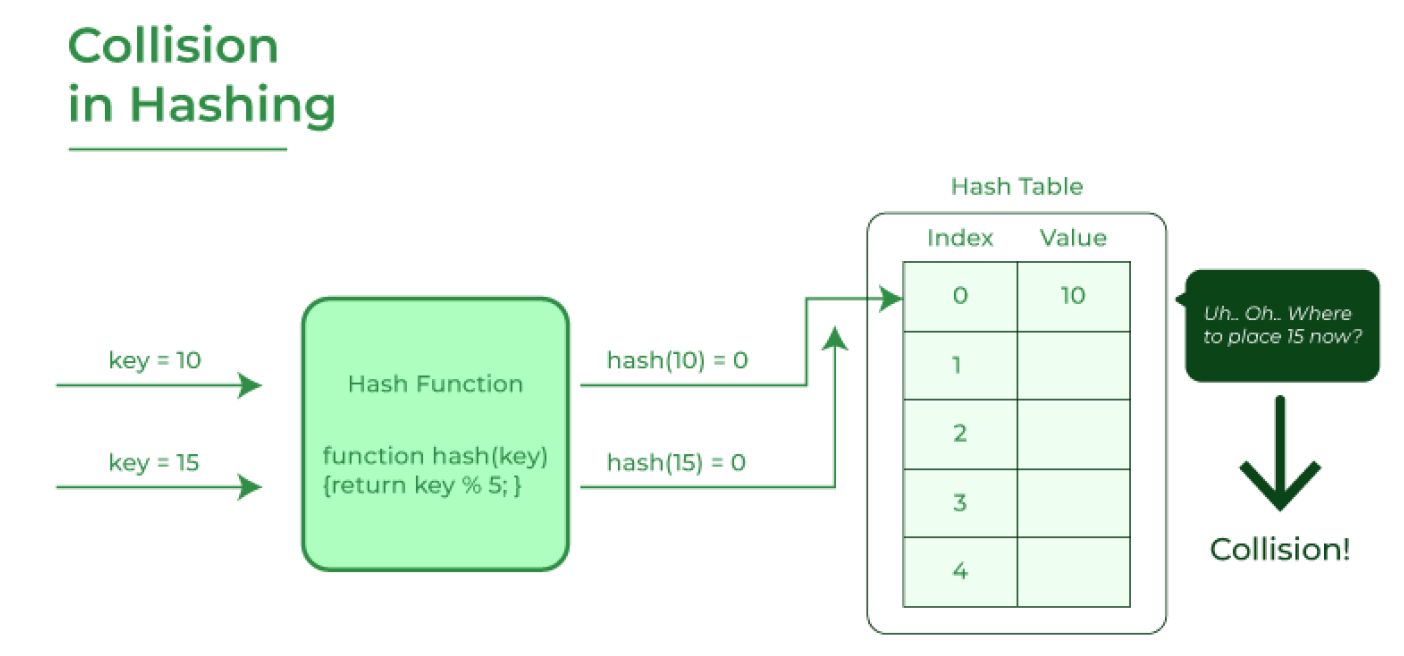
*Efficient Search:* Hashing allows for quick search operations, making it an ideal data structure for applications that require frequent search operations.

*Space-Efficient:* Hashing can be more space-efficient than other data structures, as it only requires a fixed amount of memory to store the hash table.

***Limitations of Hashing:***

*Hash Collisions:* Hashing can produce the same hash value for different keys, leading to hash collisions. To handle collisions, we need to use collision resolution techniques like chaining or open addressing.

*Hash Function Quality:* The quality of the hash function determines the efficiency of the hashing algorithm. A poor-quality hash function can lead to more collisions, reducing the performance of the hashing algorithm.



***Static Hashing***

In static hashing, the size of the hash table is fixed and does not change after it is created. The number of buckets (slots) is determined in advance, and this size remains constant, even if the number of elements in the hash table grows beyond its capacity.

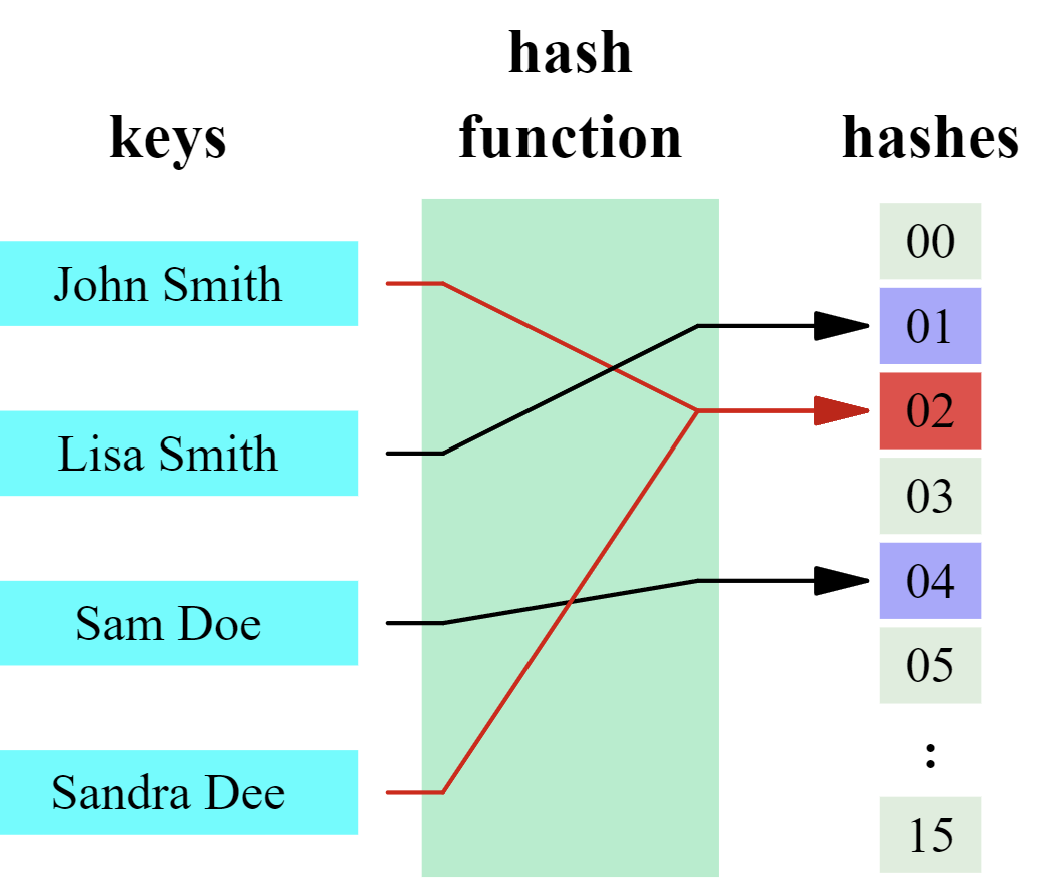
*Characteristics:*

Fixed Size: The table has a predefined size, which must be chosen carefully to avoid frequent collisions.

Simple: Since the size of the hash table is constant, implementation is straight forward.

Collisions: When the hash table becomes full or has a high load factor, collisions increase. Various collision-handling techniques like chaining (using linked lists) or open addressing (probing) are used.

No Resizing: Once the table is created, it cannot grow or shrink. This limits its flexibility in handling varying data sizes.



*What is collision in hashing?*

If we insert an element say 15 to existing hash table

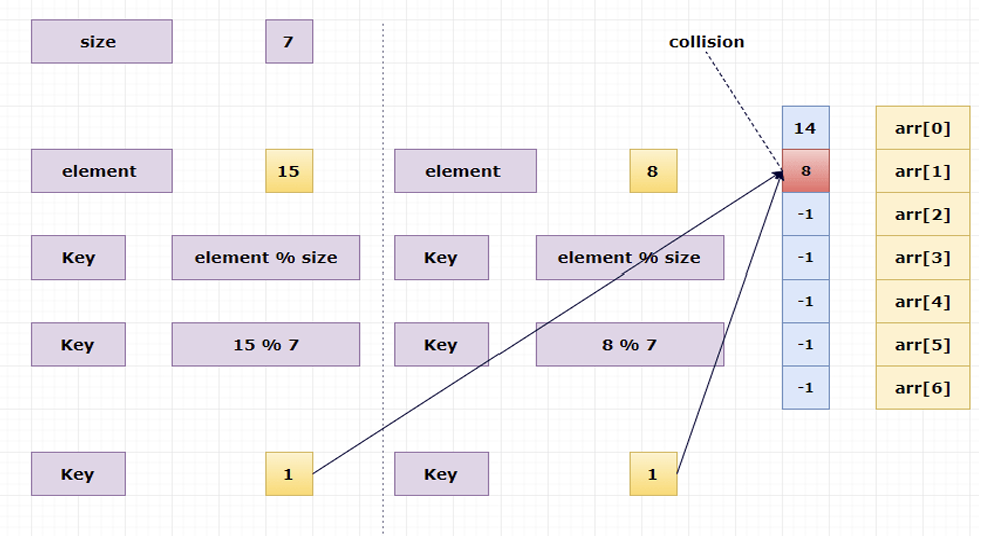
Insert : 15

Key = element % size

Key = 15%7

Key =1

But already arr[1] has element 8



*Advantages of Static Hashing*

(1) It is simple to implement.

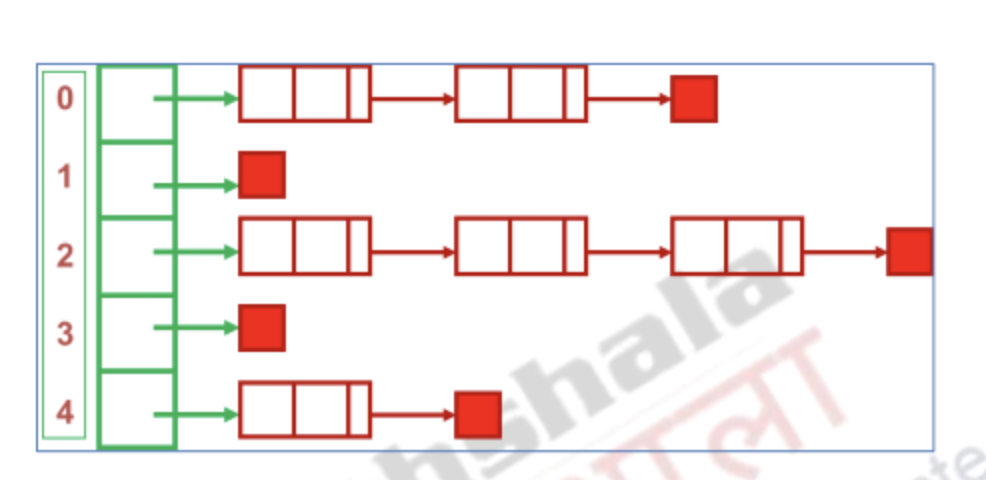
(2) It allows speedy data storage.

*Disadvantages of Static Hashing*

There are two major disadvantages of static hashing:

1) In static hashing, there are fixed number of buckets. This will create a problematic situation if the number of records grow or shrink.

2) The ordered access on hash key makes it inefficient.



*Limitations:*

Poor memory utilization if the table is under-utilized (low load factor).

High collision rate when the table is nearly full (high load factor).

It doesn’t support dynamic scaling when more data is added.

Here two or more different elements pointing to the same index under modulo size.

This is called collision

#include<stdio.h>

#define size 7

int arr[size];

void init()

{

int i;

for(i = 0; i < size; i++)

arr[i] = -1;

}

void insert(int value)

{

int key = value % size;

if(arr[key] == -1)

{

arr[key] = value;

printf("%d inserted at arr[%d]\n", value,key);

}

else

{

printf("arr[%d] has element %d already!\n",key,arr[key]);

printf("Unable to insert %d\n",value);

}

}

void del(int value)

{

int key = value % size;

if(arr[key] == value)

arr[key] = -1;

else

printf("%d not present in the hash table\n",value);

}

void search(int value)

{

int key = value % size;

if(arr[key] == value)

printf("Search Found\n");

else

printf("Search Not Found\n");

}

void print()

{

int i;

for(i = 0; i < size; i++)

printf("arr[%d] = %d\n",i,arr[i]);

}

int main()

{

init();

insert(10); //key = 10 % 7 ==> 3

insert(4); //key = 4 % 7 ==> 4

insert(2); //key = 2 % 7 ==> 2

insert(3); //key = 3 % 7 ==> 3 (collision)

printf("Hash table\n");

print();

printf("\n");

printf("Deleting value 10..\n");

del(10);

printf("After the deletion hash table\n");

print();

printf("\n");

printf("Deleting value 5..\n");

del(5);

printf("After the deletion hash table\n");

print();

printf("\n");

printf("Searching value 4..\n");

search(4);

printf("Searching value 10..\n");

search(10);

return 0;

}

***Dynamic Hashing***

Dynamic hashing allows the hash table to grow or shrink as the number of elements changes. It overcomes the limitations of static hashing by adjusting the size of the hash table dynamically to

maintain an optimal load factor.

*Characteristics:*

Resizable: The hash table can grow (expand) or shrink (contract) based on the number of elements.

Reduced Collisions: As the table size adjusts dynamically, it reduces the chances of collisions as the number of elements increases.

Memory Efficiency: Since the table adjusts its size based on the number of elements, it uses memory more efficiently.

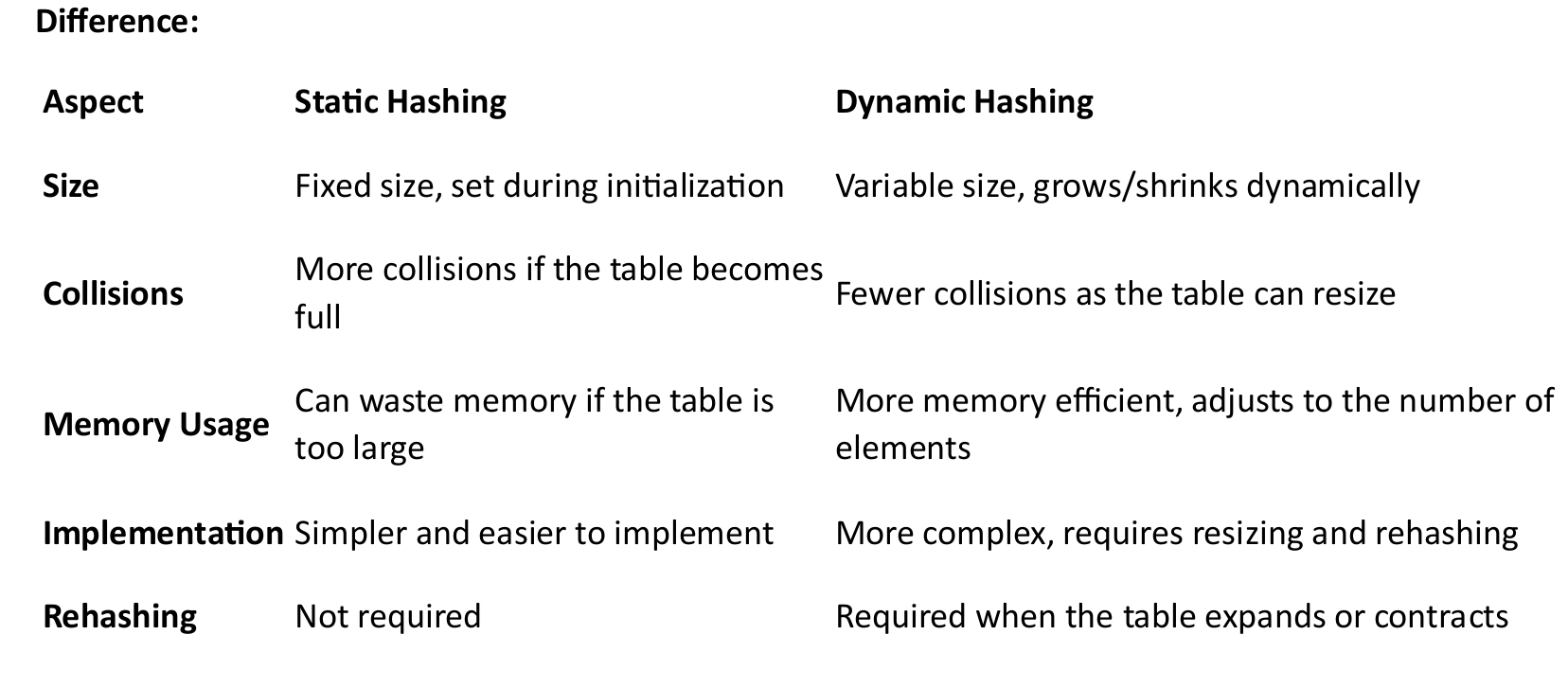
More Complex: Dynamic hashing is more complex to implement compared to static hashing because it involves table resizing and rehashing of elements when the table expands or contracts.

*Dynamic Hashing Techniques:*

Rehashing: When the load factor exceeds a certain threshold (e.g., 0.7), the table is resized (usually doubled) and all elements are rehashed (inserted into the new table based on the new hash function).

Extendible Hashing: A form of dynamic hashing that uses a directory to manage pointers to different hash tables that can grow independently.

Linear Hashing: Allows the hash table to grow gradually without the need to rehash the entire table at once.



*Advantages of Hashing in Data Structure*

Key-value support: Hashing is ideal for implementing key-value data structures.

Fast data retrieval: Hashing allows for quick access to elements with constant-time complexity.

Efficiency: Insertion, deletion, and searching operations are highly efficient.

Memory usage reduction: Hashing requires less memory as it allocates a fixed space for storing

elements.

Scalability: Hashing performs well with large data sets, maintaining constant access time.

Security and encryption: Hashing is essential for secure data storage and integrity verification.

**Open Addressing Techniques**

Three techniques are commonly used to compute the probe sequence required for open addressing:

Linear Probing.

Quadratic Probing.

Double Hashing.

*1. Linear Probing:*

It is a Scheme in Computer Programming for resolving collision in hash tables.

Suppose a new record R with key k is to be added to the memory table T but that the memory locations with the hash address H (k). H is already filled.

Our natural key to resolve the collision is to crossing R to the first available location following T (h). We assume that the table T with m location is circular, so that T [i] comes after T [m].

The above collision resolution is called "Linear Probing".

Linear probing is simple to implement, but it suffers from an issue known as primary clustering. Long runs of occupied slots build up, increasing the average search time. Clusters arise because an empty slot proceeded by i full slots gets filled next with probability (i + 1)/m. Long runs of occupied slots tend to get longer, and the average search time increases.

Given an ordinary hash function h': U {0, 1...m-1}, the method of linear probing uses the hash function.

**h (k, i) = (h' (k) + i) mod m**

Where 'm' is the size of hash table and **h' (k) = k mod m**. for i=0, 1....m-1.

Given key k, the first slot is T [h' (k)]. We next slot T [h' (k) +1] and so on up to the slot T [m-1]. Then we wrap around to slots T [0], T [1]....until finally slot T [h' (k)-1]. Since the initial probe position dispose of the entire probe sequence, only m distinct probe sequences are used with linear probing.

*2. Quadratic Probing:*

Suppose a record R with key k has the hash address H (k) = h then instead of searching the location with addresses h, h+1, and h+ 2...We linearly search the locations with addresses

h, h+1, h+4, h+9...h+i2

Quadratic Probing uses a hash function of the form

h (k,i) = (h' (k) + c1i + c2i2) mod m

Where (as in linear probing) h' is an auxiliary hash function c1 and c2 ≠0 are auxiliary constants and i=0, 1...m-1. The initial position is T [h' (k)]; later position probed is offset by the amount that depend in a quadratic manner on the probe number i.

*3. Double Hashing:*

Double Hashing is one of the best techniques available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations.

Double hashing uses a hash function of the form

h (k, i) = (h1(k) + i h2 (k)) mod m

Where h1 and h2 are auxiliary hash functions and m is the size of the hash table.

h1 (k) = k mod m or h2 (k) = k mod m'. Here m' is slightly less than m (say m-1 or m-2).