CITY GOVERNEMENT OF SAN PABLO
DALUBHASAAN NG LUNSOD NG SAN PABLO
CHED Recognized Local College
TESDA Recognized Programs
ALCU Commission on Accreditation – Level 1 Reaccredited
Member, Association of Local Colleges and Universities
Member, Local Colleges and Universities Athletic Association, Inc.

Excellence • Leadership • Service

MODULE

Data Structures and Algorithm

CC214

ACADEMIC YEAR 2024-2025

Prepared by:

BRYAN MAY Q. BOONGALING
Course Instructor

MODULE FOR DATA STRUCTURES AND ALGORITHM

Credits : **3 Units** (3 Hours Laboratory, 2 Hours Lecture)

Pre-Requisite : **CC123 – Computer Programming 2**

Lesson Title:

Lesson 7 – Hashing Algorithm

A. Concept of Hashing
B. Components of Hashing
C. Collision Resolution Techniques
D. Application of Hashing Algorithm
E. Implementation of Hashing Algorithm (C++)

Lesson Objective:

At the end of the module, the learners will be able to:

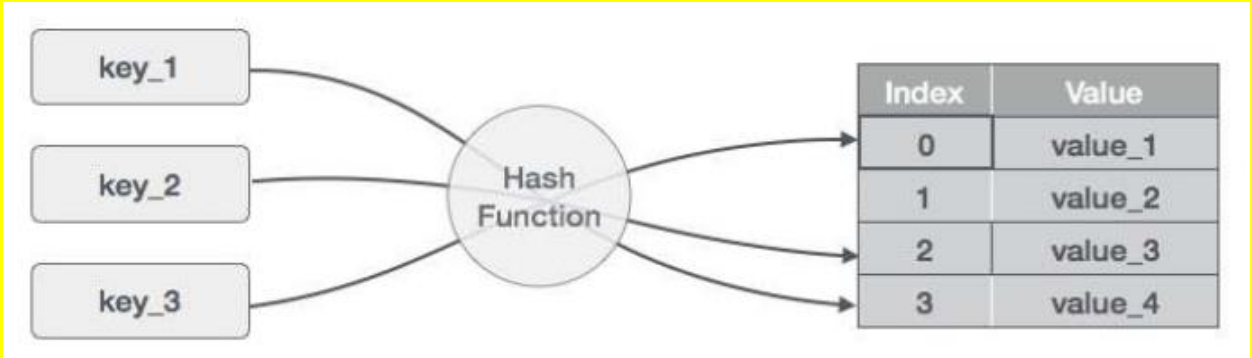
1. Explore the ideas and concepts of Hashing Algorithm relative to Data Structures.
2. Apply Hashing Techniques into computer – based and coding problem.
3. Compare and contrast the procedures used on each Hashing Techniques.
4. Provide real – world examples of Hashing Algorithms from daily life activities.

Lectures and Annotations:

Annotation:

Hashing is designed to solve the problem of needing to efficiently find or store an item in a collection. Hashing is a technique to make things more efficient by effectively narrowing down the search at the outset.

LESSON 5 – HASHING ALGORITHM



(https://www.geeksforgeeks.org/what-is-hashing/)

Hashing is an important Data Structure which is designed to use a special function called the **Hash Function** which is used to map a given value with a particular key for faster access of elements. The efficiency of mapping depends of the efficiency of the Hash Function used.

Hash Table uses an Array as a storage medium and uses Hash Technique to generate an index where an element is to be inserted or is to be located from.

A. CONCEPT OF HASHING

Hashing is a technique used for storing and retrieving information as quickly as possible. It is used to perform optimal searches and is useful in implementing symbol tables.

Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects. Some examples of how Hashing is used in our lives include:

- In Universities, each Student is assigned a unique roll number that can be used to retrieve information about them. (Student No. ID)
- In Libraries, each Book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc. (Book No. / ID)
- In both these examples the Students and Books were hashed to a "*Unique Number*".

Assume that you have an object and you want to assign a key to it to make searching easy. To store the key / value pair, you can use a simple Array like a Data Structure where keys (integers) can be used directly as an index to store values. However, in cases where the keys are large and cannot be used directly as an index, you should use **Hashing**.

B. 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 as index or location for storage of an item in a Data Structure.

2. Hash Table

Hash Table is a Data Structure which stores data in an associative manner. In a Hash Table, data is stored in an **Array Format**, where each data value has its own unique

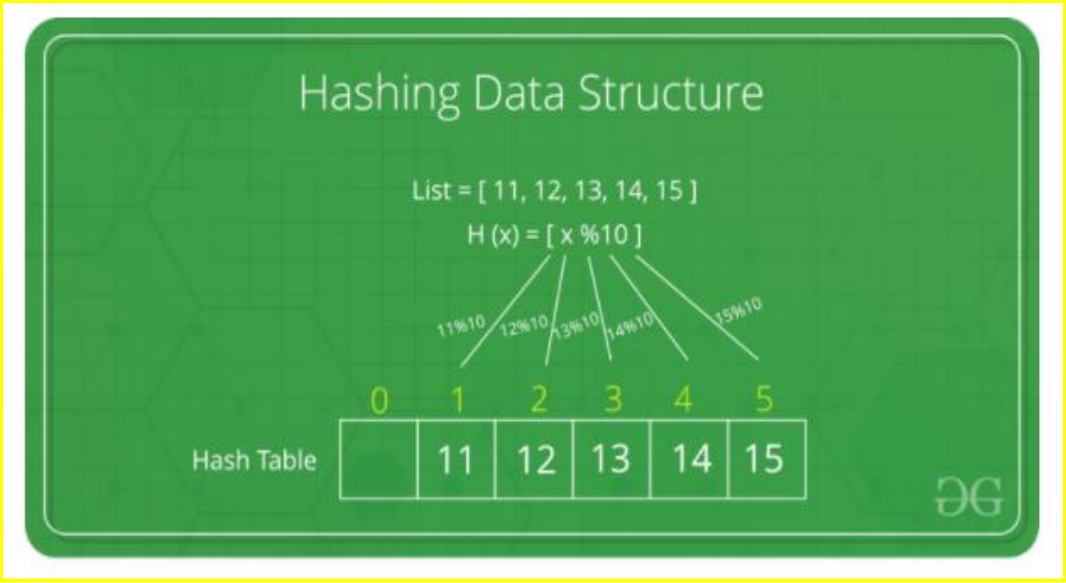
index value. Access of data becomes very fast if we know the index of the desired data. Thus, it becomes a Data Structure in which *Insertion and Search Operations* are very fast irrespective of the size of the data.

3. Hash Function

The **Hash Function** is used to transform the key into the index. Ideally, the Hash Function should map each possible key to a unique slot index, but it is difficult to achieve in practice. Given a collection of elements, a Hash Function that maps each item into a unique slot is referred to as a "*Perfect Hash Function*". If we know the elements and the collection will never change, then it is possible to construct a Perfect Hash Function.

Hashing is a technique to convert a range of key values into a range of indexes / indices of an Array. The **Modulo Operator (%)** is use to get a range of key values.

Let a **Hash Function H(x)** maps the value at the index **x % 10** in an Array. For example if the list of values is **[11, 12, 13, 14, 15]** it will be stored at positions **{1, 2, 3, 4, and 5}** in the Array or Hash Table respectively.



(*https://www.geeksforgeeks.org/hashing-data-structure/*)

Index

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
	11	**12**	**13**	**14**	**15**				

Value

EXAMPLE #1:

Problem:
Consider an example of **Hash Table** of size **20**, and the following items are to be stored. Item are in the (Key, Value) format.

Given:

Array Name: **arrayExample1**
Array Size: **20**
Hash Function: **arrayExample1 (x) = [x % 20]**

ID No.	Key / s:	Hash Function:	Array Index:
1	1	1 % 20 = 1	1
2	15	15 % 20 = 15	15
3	28	28 % 20 = 8	8
4	33	33 % 20 = 13	13
5	49	49 % 20 = 9	9
6	55	55 % 20 = 15	15
7	10	10 % 20 = 10	10
8	6	6 % 20 = 6	6
9	81	81 % 20 = 1	1
10	59	59 % 20 = 19	19
11	68	68 % 20 = 8	8
12	20	20 % 20 = 0	0

Hash Table:

Index

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*
20	**1 81**					**6**		**28 68**	**49**	**10**			**33**		**15 55**				**59**

Value

Collision:

Collision Occurs at Index {1, 8, 15}

- arrayExample1 [1]
- arrayExample1 [8]
- arrayExample1 [15]

C. COLLISION RESOLUTION TECHNIQUES

What is Collision?

Hash Functions are used to map each key to a different address space, but practically it is not possible to create such a Hash Function and the problem is called **Collision**. Collision is the condition where two records / data are stored in the same location / address.

Collision Resolution Scheme / Techniques:

The process of finding an alternate location is called **Collision Resolution**. Even though Hash Tables have collision problems, they are more efficient in many cases compared to all other Data Structures, like *Search Trees*.

There are a number of Collision Resolution Techniques, and the most popular are *Direct Chaining* and *Open Addressing*.

1. **Direct Chaining**: An Array of Linked List Application
    - Separate Chaining Algorithm

2. **Open Addressing**: Array – Based Implementation
    - Linear Probing Algorithm (Linear Search)
    - Quadratic Probing Algorithm (Non – Linear Search)
    - Double Hashing Algorithm (Use Two Hash Functions)

I. LINEAR PROBING ALGORITHM

As we can see, it may happen that the Hashing Technique is used to create an already used index of the Array. In such a case, we can search the next empty location in the Array by looking into the next cell until we find an empty cell.

A **Linear Probing** is an array – based implementation that uses a simple Re – Hashing Scheme in which the next slot in the table is checked on a Collision.

EXAMPLE #2:

Problem:
Consider an example of **Hash Table** of size **20**, and the following items are to be stored. Item are in the (Key, Value) format.

Given:

Array Name: **arrayExample2**
Array Size: **20**
Hash Function: **arrayExample2 (x) = [x % 20]**

ID No.	Key / s:	Hash Function:	Array Index:	Array Index: (After Linear Probing)
1	1	1 % 20 = 1	1	1
2	15	15 % 20 = 15	15	15
3	28	28 % 20 = 8	8	8
4	33	33 % 20 = 13	13	13
5	49	49 % 20 = 9	9	9
6	55	55 % 20 = 15	15	16
7	10	10 % 20 = 10	10	10
8	6	6 % 20 = 6	6	6
9	81	81 % 20 = 1	1	2
10	59	59 % 20 = 19	19	19
11	68	68 % 20 = 8	8	11
12	20	20 % 20 = 0	0	0

Hash Table:

Index

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*
20	**1**	**81**				**6**		**8**	**49**	**10**	**68**		**13**		**15**	**55**			**59**

Value

Collision:

Collision Occurs at Index {1, 8, 15}

- **arrayExample2 [1] ->** re – hashed to **arrayExample2 [2]**
- **arrayExample2 [8] ->** re – hashed to **arrayExample2 [11]**
- **arrayExample2 [15] ->** re – hashed to **arrayExample2 [16]**

Solution:

6. 2
20 / 55
- 40
15 - - - - - Collision Occurs at arrayExample2 [15]

First Probe: *(i = 1)* 15 + (1) = 16 - - - - - Empty Slot, Insert at arrayExample2 [16]

9. 4
20 / 81
- 80
1 - - - - - Collision Occurs at arrayExample2 [1]

First Probe: *(i = 1)* 1 + (1) = 2 - - - - - Empty Slot, Insert at arrayExample2 [2]

11. 3
20 / 68
- 60
8 - - - - - Collision Occurs at arrayExample2 [8]

First Probe: *(i = 1)* 8 + (1) = 9 - - - - - Collision Occurs at arrayExample2 [9]

Second Probe: *(i = 2)* 8 + (2) = 10 - - - - - Collision Occurs at arrayExample2 [10]

Third Probe: *(i = 3)* 8 + (3) = 11 - - - - - Empty Slot, Insert at arrayExample2 [11]

II. QUADRATIC PROBING ALGORITHM

Quadratic Probing is a collision resolution technique used in open addressing methods for hash tables. When a hash table is full or a hash function results in a collision (*i.e., two keys hash to the same index*), Quadratic Probing helps find the next available slot using a quadratic function to compute the probe sequence.

Instead of probing linearly, Quadratic Probing uses the following formula to calculate the next index to check: **h (k, i) = (h (k) + i^2) mod m,** where:

- ***h(k)*** is the hash value of the key ***k*** (the initial hash index)
- ***i*** is the number of probes (attempts to find an empty slot)
- ***m*** is the size of the hash table
- The quadratic function is applied as ***i^2*** to generate successive indices

EXAMPLE #3:

Problem:
Consider an example of **Hash Table** of size **20**, and the following items are to be stored. Item are in the (Key, Value) format.

Given:

Array Name: **arrayExample3**
Array Size: **20**
Hash Function: **arrayExample3 (x) = [x % 20]**

ID No.	Key / s:	Hash Function:	Array Index:	Array Index: (After Quadratic Probing)
1	1	1 % 20 = 1	1	1
2	15	15 % 20 = 15	15	15
3	28	28 % 20 = 8	8	8
4	33	33 % 20 = 13	13	13
5	49	49 % 20 = 9	9	9
6	55	55 % 20 = 15	15	16
7	10	10 % 20 = 10	10	10
8	6	6 % 20 = 6	6	6
9	81	81 % 20 = 1	1	2
10	59	59 % 20 = 19	19	19
11	68	68 % 20 = 8	8	12
12	20	20 % 20 = 0	0	0

Hash Table:

Index

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*
20	**1**	**81**				**6**		**28**	**49**	**10**		**68**	**13**		**15**	**55**			**59**

Value

Collision:

Collision Occurs at Index {1, 8, 15}

- **arrayExample3 [1] ->** re – hashed to **arrayExample3 [2]**
- **arrayExample3 [8] ->** re – hashed to **arrayExample3 [12]**
- **arrayExample3 [15] ->** re – hashed to **arrayExample3 [16]**

Solution: Formula: **$h(k, i) = (h(k) + i^2) \bmod m$**

6. 2
20 / 55
- 40
15 - - - - - Collision Occurs at arrayExample3 [15]

First Probe: *(i = 1)* 15 + (1 * 1) = 16 - - - - - Empty Slot, Insert at arrayExample3 [16]

9. 4
20 / 81
- 80
1 - - - - - Collision Occurs at arrayExample3 [1]

First Probe: *(i = 1)* 1 + (1 * 1) = 2 - - - - - Empty Slot, Insert at arrayExample3 [2]

11. 3
20 / 68
- 60
8 - - - - - Collision Occurs at arrayExample3 [8]

First Probe: *(i = 1)* 8 + (1 * 1) = 9 - - - - - Collision Occurs at arrayExample3 [9]

Second Probe: *(i = 2)* 8 + (2 * 2) = 12 - - - - - Empty Slot, Insert at arrayExample3 [12]

D. APPLICATION OF HASHING ALGORITHM

Hashing Algorithms are widely used in Computer Science and Software Engineering due to their efficiency in solving various problems related to *Searching, Indexing, and Organizing Data*. Below are several important applications of Hashing Algorithms:

1. **Hash Table (Associative Arrays)**
   - Storing key – value pairs for fast retrieval.
   - Example: Implementing dictionaries or maps in programming languages like **Python** (dict), **Java** (HashMap), or **C++** (unordered_map).

2. **Data Indexing**
   - Efficient searching and retrieval in databases.
   - Example: Hash – based indexing in RDBMS such as **MySQL** and **PostgreSQL**.

3. **Password Storage**
   - Securely storing passwords in databases.
   - Example: Storing passwords using **bcrypt**, **SHA-256**, or **PBKDF2** hash functions in web applications (e.g., user authentication in websites).

4. **Cryptography**
   - Creating Digital Signatures and Message Authentication Codes (MAC).
   - Example: **SHA-256** is used in the Bitcoin Blockchain for generating the hash of each block in the chain.

5. **Data Integrity Checks**
   - Verifying data integrity during transmission.
   - Example: **MD5**, **SHA-1**, or **SHA-256** hashes are commonly used to verify the integrity of files downloaded from the internet.

6. **Caching**
   - Speeding up retrieval of frequently accessed data.
   - Example: Caching database queries, web pages, or function calls in distributed systems like **Memcached** or **Redis**.

7. **Load Balancing**
   - Distributing tasks or requests across multiple servers.
   - Example: Consistent hashing is used in systems like **Amazon DynamoDB**, **HashiCorp Consul**, or **Cassandra** to distribute data evenly across multiple servers.

8. **Duplicate Detection**
   - Identifying duplicate data in large database .
   - Example: Content – based deduplication in cloud storage systems like **Dropbox** or **Google Drive**.

9. **Distributed Systems (Data Partitioning)**
   - Partitioning data across multiple nodes in a distributed system.
   - Example: Consistent hashing is used to efficiently distribute data in systems like **Cassandra**, **Amazon DynamoDB**, or **HBase**.

10. **File or Data Fingerprints**
    - Identifying unique data based on content.
    - Example: **Git** uses hashing (e.g., **SHA-1**) to track versions of files in a version control system, allowing it to identify and store changes efficiently.

11. **Random Access in File Systems**
    - Quick access to specific locations in files or databases.
    - Example: The **FAT** and **NTFS** uses hashing techniques for quick file lookup and storage management.

12. **Digital Fingerprints in Media**
    - Verifying and tracking digital media.
    - Example: **YouTube's Content ID** uses hashing to detect uploaded videos that match copyrighted content.

13. **Bloom Filters**
    - Testing membership in a set with minimal memory usage.
    - Example: **Web Crawling** to quickly check if a URL has been visited before, or in databases to filter out non-matching queries before performing full scans.

References:

Chaudhuri, A. B. (2020). Flowchart and Algorithm Basics: The Art of Programming. Mercury Learning and Information.

Malik, D. S. (2019). C++ Programming Including Data Structures. Cengage Learning,

Puntambekar, A. A. (2019). Data Structures Usisng C: A Conceptual Approach (1st Edition). Technical Publications.

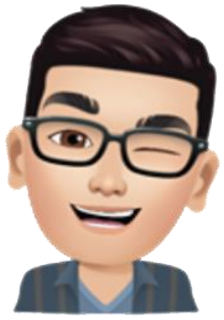
Karumanchi, N. (2017). Data Structures and Algorithms Made Easy. 5th Edition. CareerMonk Publications.

Delfinado, C. J. (2016). Data Structures and Algorithms. C & E Publishing, Inc.

Tutorials Point (I) Pvt. Ltd. (2016). Data Structures and Algorithms. Tutorials Point: Simply Easy Learning.

Jadeja, P. (n.d.). Introduction to Data Structure. Darshan, Institute of Engineering and Technology.

CS Dojo – YouTube Channel (2018). Data Structures and Algorithms (Videos 1 - 13). Retrieved from https://www.youtube.com/channel/UCxX9wt5FWQUAAz4UrysqK9A



XDXDXDXDXDXDXDXDXDXDXDXDXDXD