Course title	: **CSE2001**
Course title	: **Data Structures and Algorithms**
Module	: **5**
Topic	: **2**

Introduction to Hashing Algorithms

Objectives

This session will give the knowledge about

- Introduction to Hashing Algorithms
- Types of Hashing
- Separate Chaining
- Linear Probe

Introduction to Hashing

In all search techniques, the time required to search an element depends on the total number of elements.

Hashing is another approach in which time required to search an element doesn't depend on the total number of elements.

Using hashing data structure, a given element is searched with constant time complexity.

Definition

Hashing is the process of indexing and retrieving element (data) in a data structure to provide a faster way of finding the element using a hash key.

Here, the hash key is a value which provides the index value where the actual data is likely to be stored in the data structure.

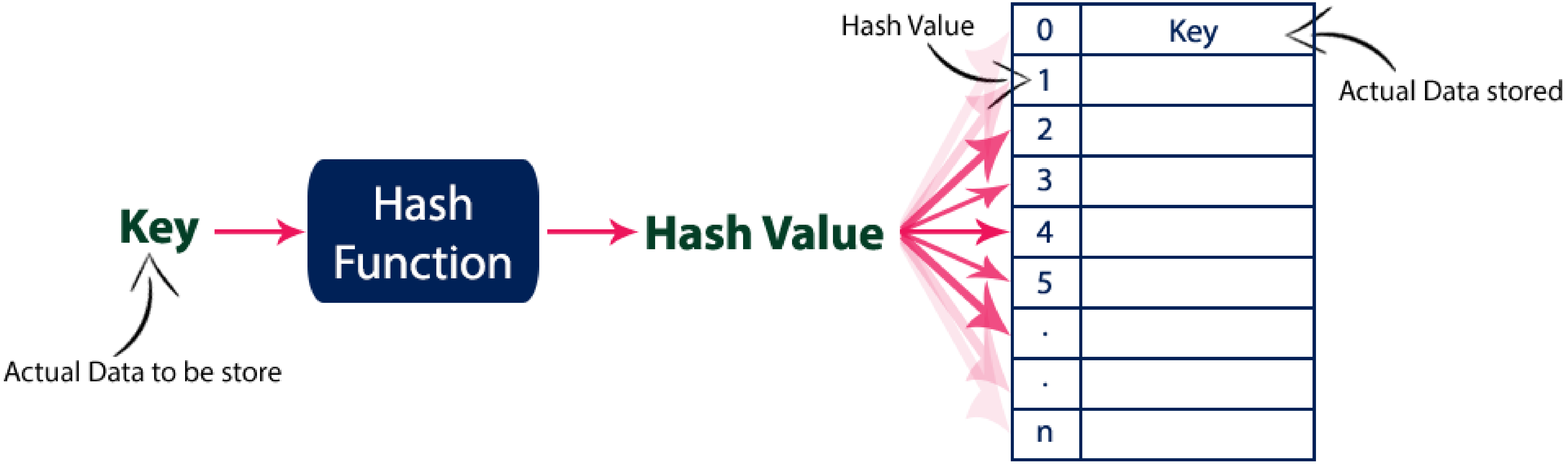
Hash Table

Hash Table is a data structure which stores data in an associative manner.

- All the data values are inserted into the hash table based on the hash key value.
- The hash key value is used to map the data with an index in the hash table.
- And the hash key is generated for every data using a hash function.

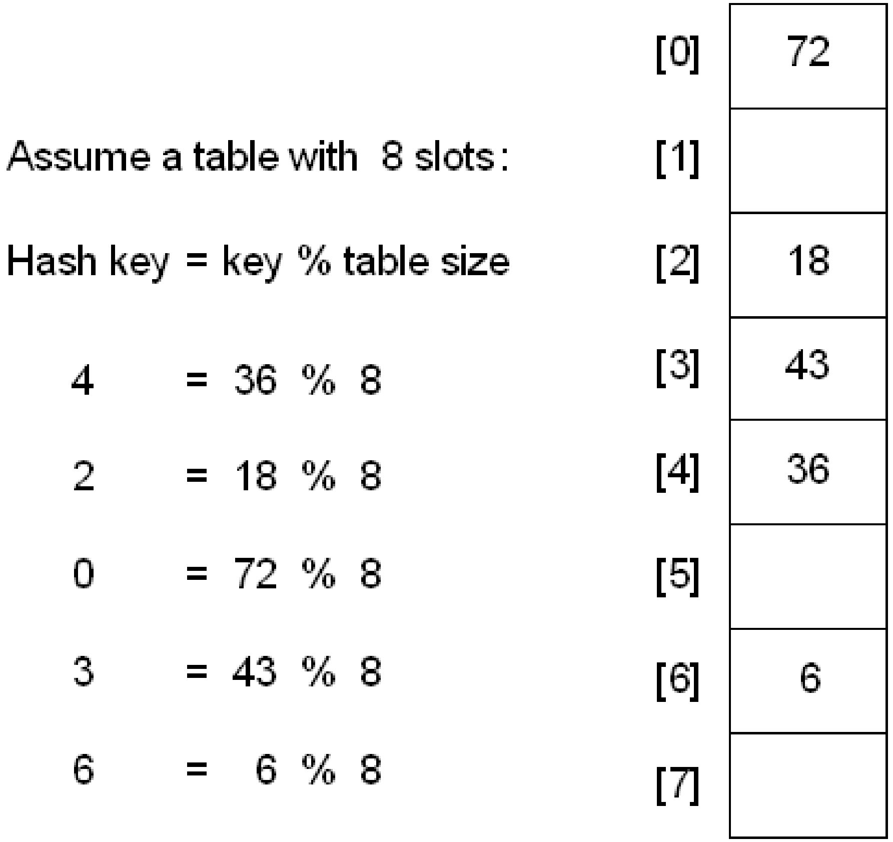
Hash function is a function which takes a piece of data (i.e. key) as input and produces an integer (i.e. hash value) as output which maps the data to a particular index in the hash table.

Basic concept of hashing and hash table



Hashing

hash key = key % number of slots in the table



Assume a table with 8 slots:

Hash key = key % table size

4 = 36 % 8

2 = 18 % 8

0 = 72 % 8

3 = 43 % 8

6 = 6 % 8

[0]	72
[1]	
[2]	18
[3]	43
[4]	36
[5]	
[6]	6
[7]	

Hashing – Creating Hash table

```
int[] arr;
int capacity;

/** constructor **/
public HashTable(int capacity)    {
    this.capacity = capacity;
    arr = new int[this.capacity];
}
```

Hashing - Insert

```
/** function to insert **/
public void insert(int ele)
{
    arr[ele % capacity] = ele;
}

/** function to clear **/
public void clear()
{
    arr = new int[capacity];
}
```

Hashing - Remove

```
 /** function contains **/
public boolean contains(int ele)    {
    return arr[ele % capacity] == ele;
}

/** function to delete **/
public void delete(int ele)
{
    if (arr[ele % capacity] == ele)
        arr[ele % capacity] = 0;
    else
        System.out.println("\nError : Element not found\n");
}
```

Hashing – Display Hash table

```
/** function to print hash table **/
public void printTable()
{
    System.out.print("\nHash Table = ");
    for (int i = 0; i < capacity; i++)
        System.out.print(arr[i] +" ");
    System.out.println();
}
```

Types of Hashing

Depends on the datatype we are storing on the hash table, different types of hashing can be applied. Some important methods are:

- Modulo division - hf(k) = key % 10 or hf(k) = key %n
- Mid square - hf(k) = returns mid value of key * key
- Digit extraction - Extract some digits from key
- Digit folding – Divide key into n parts and sum
- Multiplication method - (h(k) = floor(n(kA mod 1)), A=0.618033)

Modulo division method

Modulo division - hf(k) = key % 10 hf(k) = key %n

Assume Hash Table size is = 5

20, 23, 34, 22

hf(20) = 20 % 5 = 0

hf(23) = 23 % 5 = 3

hf(34) = 34 % 5 = 4

hf(29) = 22 % 5 = 2

0	20
1	
2	22
3	25
4	34

Mid Square method

Mid square - hf(k) = returns mid value of key * key

Assume Hash table size is 10

27, 32, 41, 38

$hf(27) = 27 \times 27 = \cancel{7}2\cancel{9} = 2$

$hf(32) = 32 \times 32 = \cancel{1}\underline{02}\cancel{4}$

↳ take left => 0

$hf(41) = 41 \times 41 = \cancel{1}\underline{68}\cancel{1}$ => take left => 6

$hf(38) = 38 \times 38 = \cancel{1}44\cancel{4}$ => take left => 4

0	32
1	
2	27
3	
4	38
5	
6	41
7	
8	
9	

Digit Extraction method

Digit extraction - Extract some digits from key

Assume Hash Table size is 10

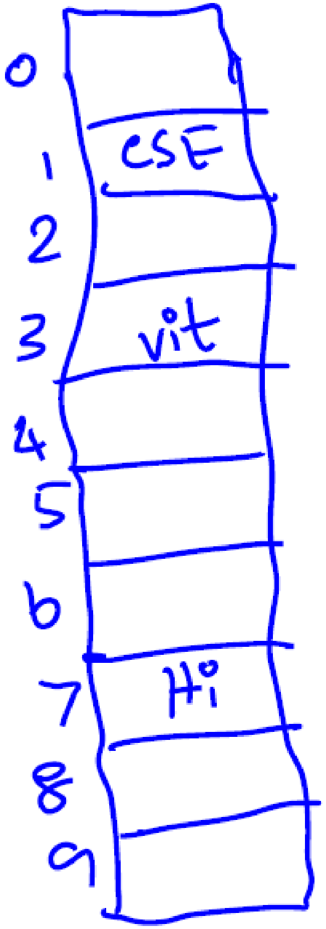
Hi, vit, CSE -> find sum of ASCII values

Hi -> 72+105 -> 177 => hf(Hi) -> 7

vit -> 118+105+116 -> 339 => hf(vit) -> 3

CSE -> 67+83+69 -> 219 => hf(CSE) -> 1

here HT size is single digit. Hence I need to extract single digit (2nd digit)



0	
1	CSE
2	
3	vit
4	
5	
6	
7	Hi
8	
9	

Digit folding method

Digit folding – Divide key into n parts and sum

Assume HashTable size is 100

734523, 7321, 71645

6/2 ⇒ 3 hence divide i/p into 3 parts

hf(734523) = 73 + 45 + 23 = 141 ⇒ 41

omit ↩ 1

hf(7321) ⇒ 73 | 21 | 0

→ 73 + 21 = 94

hf(71645) ⇒ 71 + 64 + 5 = ~~1~~40

Hash Table

0

⋮

99

⇒

40 → 71645

41 → 734523

94 → 7321

Multiplication method

Multiplication method - (h(k) = floor(n(kA mod 1)), A=0.618033)

Assume Hashtable size is $\underline{\underline{10}}$

7, 8, 4, 3

$hf(7) = \text{floor}(10 * ((\underbrace{7 * 0.618033)}_{4.326 \% 1} \% 1)$

$\underline{10 \times 0.326}$

floor (3.26) $\Rightarrow$ 3

$hf(8) = \underline{4.94664 \% 1}$

0.94664 *10 $\Rightarrow$ 9.4664 $\Rightarrow$ floor(9.4) $\Rightarrow$ 9

hash Table

Index		value
0		
1		
2		
3	→	7
:		
9	→	8

Hashing - Collision

The previous method is simple, but it is flawed if the table size is large. For example, assume a table size of 10007 and that all keys are eight or fewer characters long.

No matter what the hash function, there is the possibility that two keys could resolve to the same hash key. This situation is known as a collision.

When this occurs, there are two simple solutions:

- chaining
- linear probe (aka linear open addressing)

And two slightly more difficult solutions

- Quadratic Probe
- Double Hashing

Types of Hashing Collision Resolution

If the hash function generates same hash index for different keys, collision will occur. Some important Collision resolution methods are:

- Separate Chaining - Implement hash table as LinkedHashTable
  - Never overflow
  - No need for rehashing

- Linear Probing – Next Possible position

Types of Hashing Collision Resolution

- Quadratic Probing - $(h(k)+ i^2)$ mod N
  - Where i is the occurance of the collosion for the given key

- Double Hashing - R - (key % R) where R is a prime number that is smaller than the size of the table. Further, (hash1(key) + i * hash2(key)) % TABLE_SIZE
  - Double hashing can be applied only if the table size is prime number

Types of Hashing Collision Resolution

If the hash function generates same hash index for different keys, collision will occur. Some important Collision resolution methods are:

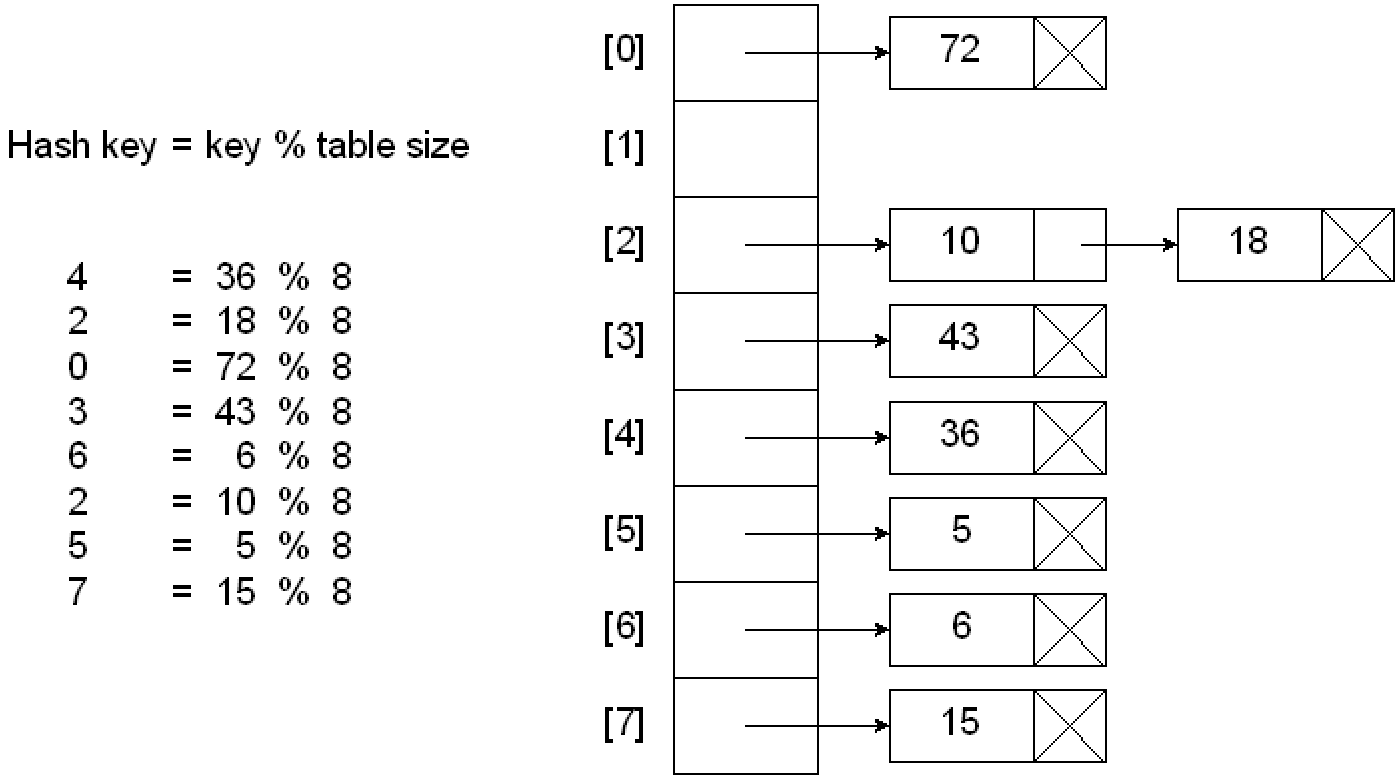
- Separate Chaining - Implement hash table as LinkedHashTable
- Linear Probing – Next Possible position
- Quadratic Probing - $(h(k)+ i^2)$ mod N
- Double Hashing - R - (key % R) where R is a prime number that is smaller than the size of the table. Further, (hash1(key) + i * hash2(key)) % TABLE_SIZE
- Re hashing – Multiply hash table size by 2 and find next prime number. Then continue hashing from first

Types of Hashing Collision Resolution

- Re hashing – Multiply hash table size by 2 and find next prime number. Then continue hashing from first
  - Rehashing will be called when load factor is more than the threshold level
  - Load factor will be calculated as number of filled places / table size
  - Threshold level can be user defined. Suggestable is 0.7

Hashing with Chains (Separate Chaining)

When a collision occurs, elements with the same hash key will be chained together. A chain is simply a linked list of all the elements with the same hash key.



Hash key = key % table size

4 = 36 % 8
2 = 18 % 8
0 = 72 % 8
3 = 43 % 8
6 = 6 % 8
2 = 10 % 8
5 = 5 % 8
7 = 15 % 8

Separate Chaining – Creating hash object

```
/* Class LinkedHashEntry */
class LinkedHashEntry  {
        String key;
        int value;
        LinkedHashEntry next;
        /* Constructor */
        LinkedHashEntry(String key, int value)    {
                this.key = key;
                this.value = value;
                this.next = null;
        }
}
```

Separate Chaining – Creating hash table

```
private int TABLE_SIZE;
private int size;
private LinkedHashEntry[] table;

/* Constructor */
public HashTable(int ts)  {
    size = 0;
    TABLE_SIZE = ts;
    table = new LinkedHashEntry[TABLE_SIZE];
    for (int i = 0; i < TABLE_SIZE; i++)
        table[i] = null;
}
```

Separate Chaining – Insert

```
/* Function to insert a key value pair */
public void insert(String key, int value)    {
    int hash = (myhash( key ) % TABLE_SIZE);
    if (table[hash] == null)
           table[hash] = new LinkedHashEntry(key, value);
    else {
           LinkedHashEntry entry = table[hash];
        while (entry.next != null && !entry.key.equals(key))
                            entry = entry.next;
        if (entry.key.equals(key))
                            entry.value = value;
        else
            entry.next = new LinkedHashEntry(key, value);
    }
    size++;
}
```

Separate Chaining – Remove

```
public void remove(String key)   {
        int hash = (myhash( key ) % TABLE_SIZE);
        if (table[hash] != null)    {
            LinkedHashEntry prevEntry = null;
            LinkedHashEntry entry = table[hash];
            while (entry.next != null && !entry.key.equals(key))  {
                prevEntry = entry;
                entry = entry.next;
            }
            if (entry.key.equals(key))  {
                if (prevEntry == null)
                    table[hash] = entry.next;
                else
                    prevEntry.next = entry.next;
                size--;
            }        }     }
```

Hashing with Linear Probe

When using a linear probe, the item will be stored in the next available slot in the table, assuming that the table is not already full.

[0]	72
[1]	
[2]	18
[3]	43
[4]	36
[5]	
[6]	6
[7]	

Add the keys 10, 5, and 15 to the previous table .

Hash key = key % table size

2 = 10 % 8

5 = 5 % 8

7 = 15 % 8

[0]	72
[1]	15
[2]	18
[3]	43
[4]	36
[5]	10
[6]	6
[7]	5

Summary

At the end of this session will learned about

- Introduction to Hashing Algorithms
- Types of Hashing
- Separate Chaining
- Linear Probe
- https://iswsa.acm.org/mphf/openDSAPerfectHashAnimation/perfectHashAV.html