

# Searching & Hashing

# General Guideline



© (2021) ABES Engineering College.

This document contains valuable confidential and proprietary information of ABESEC. Such confidential and proprietary information includes, amongst others, proprietary intellectual property which can be legally protected and commercialized. Such information is furnished herein for training purposes only. Except with the express prior written permission of ABESEC, this document and the information contained herein may not be published, disclosed, or used for any other purpose.

# Module Objective



**Linear Search – Problem – Solution –Complexity - Application - Disadvantages**

**Binary Search – Problem – Solution – Complexity-Application - Disadvantages**

**Problems – Solution And Variation of Binary Search and linear Search**

**Ternary Search – Jump Search – Exponential Search – Interpolation Search**

**Hashing – Hash Function**

**Collision Resolution Technique – Linear Probing – Quadratic Probing – Rehashing/ Double Hashing**



# Searching

# Searching :

**Searching is the process of finding a given value position in a list of values.**

**It decides whether a search key is present in the data or not.**

**It is the algorithmic process of finding a particular item in a collection of items.**

There are several methods of searching in the list of items.

Some of them are listed below:

- Linear/ Sequential Search (Brute Force approach)
- Binary Search (Decrease and conquer approach)
- Ternary Search
- Jump Search
- Exponential Search
- Indexed Sequential Search (Combination of Sequential and Binary Search)
- Hashing



# Real world Problems



There are many other scenarios in which searching is performed as a frequently carried out operation.

Some of them are listed below:

- Dictionary where meaning is required to be searched for a given the word,
- Catalog from which description is required to be searched for a particular item e.g. e-retail stores like Amazon, Flipkart
- Daily attendance through fingerprint scan (biometric search),
- Telephone directory of mobile phone in which a telephone number is required to be searched through name,
- Keyword search in Google, etc.
- Research article search at Research Gate, Sci-hub, etc.

# Linear Search



A Linear/Sequential search simply scans each element at a time sequentially; that's why it is also known as sequential search.

Example:

Suppose we have to find a mobile number for some person. The Mobile No is stored in the address book of the phone. If we scan from the first contact and scroll it down one by one until the desired mobile no is found, this process is also a sequential search.

# Algorithm

# Example

**ALGORITHM** LinearSearch(A[ ], N, SearchKey)

**BEGIN:**

FOR i=0 to N-1 DO

IF A [ i ] == SearchKey THEN

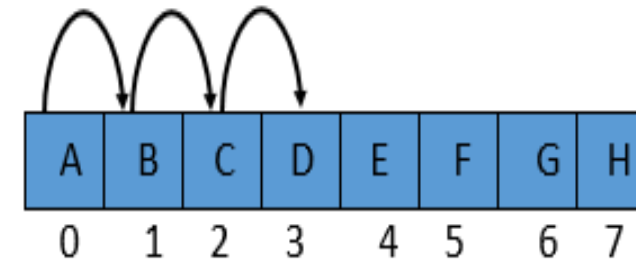
RETURN i

RETURN -1

//invalid index indicating search element is not found

**END;**

Let us consider the following example. The below-given figure shows an array of character values having 8 data items.



If we want to search 'D', then the searching begins from the 0th element and scans each element is scanned one by one till the search element is not found.



# Analysis of algorithm

## Time Complexity

**Best-case complexity:**  $\Omega(1)$ , when the element is found at the first position.

**Worst-case complexity:**  $O(n)$ , when the element is found at the last index or element is not present in the array.

**Average-case Complexity:** For the average case analysis, we need to consider the probability of finding the element at every position. In a set of randomly arranged data elements, finding the search element at any place is equally likely. In an element data set of  $n$  size, the probability of finding the element at every position will be  $1/n$ .

$$\begin{aligned}\text{Total Effort} &= \sum \text{Probability} * \text{No of Comparisons} \\ &= 1/n * 1 + 1/n * 2 + 1/n * 3 + \dots + 1/n * (n-1) + 1/n * (n) \\ &= 1/n * (1+2+3+ \dots + n) \\ &= 1/n * \sum n \\ &= 1/n * n * (n+1)/2 \\ &= (n+1)/2 \\ &= \theta(n)\end{aligned}$$

# Analysis of Algorithm



**Space Complexity:** In the algorithm written above, we just need a loop counter as additional memory. The space complexity thus is constant i.e.,  $\theta(1)$ .

# Linear Search in 2-D Array(Matrices)



**ALGORITHM LinearSearchIn2D(A[m][n], SearchKey)**

**BEGIN:**

    FOR i=0 to m-1 DO

        FOR j=0 to n-1 DO

IF A [i][j] == SearchKey THEN

                    WRITE("Element found at Row i, Column j")

    WRITE("Element not present")

**END;**

**Time complexity =  $O(n^2)$**



# Disadvantage



- In the linear search the data elements are randomly arranged.
- In case we have the data elements arranged in some order, the search effort can be brought down?
- Linear Search still takes  $O(n)$  time
- Can we Reduce ?

# Binary Search



**Binary search is an approach that can be applied ..**

**If the Binary search is performed on a sorted array, the procedure goes like:**

1. Find the middle element of the array.
2. Compare the mid element with the search element.
3. There are three possible cases.

**The mid element is the same as the search element. Search can be declared successful in this case.**

1. If the search element is less than the mid element, the search area should be restricted to the left half of mid only
2. Similarly, if it is greater than the mid element, search area should be the right half of mid.
3. The process is followed until the search element is found.

# Solution Approach

## Example:

- Let us understand the working of binary search through an example:
- Suppose we have an array of 10 size, which is indexed from 0 to 9 as shown in the below figure and we want to search element 22 (Key) in the given array.

Mid=  $\lfloor (Low+High)/2 \rfloor$  Search 22

2	4	7	11	15	22	37	55	71	90
0	1	2	3	4	5	6	7	8	9

Mid=  $\lfloor (0+9)/2 \rfloor = 4$

Search 22  
 $22 > 15$

Low=0		Mid=4			High=9				
2	4	7	11	15	22	37	55	71	90
0	1	2	3	4	5	6	7	8	9

Mid=  $\lfloor (5+9)/2 \rfloor = 7$

Search 22  
 $22 < 55$

					Low=5		Mid=7	High=9	
2	4	7	11	15	22	37	55	71	90
0	1	2	3	4	5	6	7	8	9

Mid=  $\lfloor (5+6)/2 \rfloor = 5$

Search 22  
Return 5

					Mid=5		Low=5 High=6		
2	4	7	11	15	22	37	55	71	90
0	1	2	3	4	5	6	7	8	9



# Algorithm



## ALGORITHM Binary Search (A[],N, SearchKey)

**BEGIN:**

Low=0

High = N-1

WHILE Low <=High DO

Mid =  $\lfloor (Low + High) / 2 \rfloor$

IF A[Mid] == Searchkey THEN

    RETURN Mid

ELSE

    If SearchKey < A[Mid] THEN

        High= Mid-1

    ELSE

        Low= Mid+1

RETURN -1

// invalid index indicating that element is not found

**END;**

# Analysis of Algorithm

## Time complexity:

In binary search, best-case complexity is  $\Omega(1)$ , where the element is found at the middle index in the first run.

For the worst-case analysis,

### Analysis of input size at each iteration of Binary Search:

At Iteration 1:

*Length of array =  $n$*

At Iteration 2:

*Length of array =  $n/2$*

At Iteration 3:

*Length of array =  $(n/2)/2 = n/2^2$*

Therefore, after Iteration  $k$ :

*Length of array =  $n/2^k$*

# Analysis of Algorithm

Also, we know that after  $k$  iterations, the length of the array becomes 1 Therefore, the Length of the array

$$n/2^k = 1$$

$$\Rightarrow n = 2^k$$

Applying log function on both sides:

$$\Rightarrow \log_2 n = \log_2 2^k$$

$$\Rightarrow \log_2 n = k * \log_2 2$$

$$\text{As } (\log_a (a) = 1) \text{ Therefore, } k = \log_2(n)$$

**Space Complexity:** The algorithm takes high, low, and mid variables in the logic. Count of 3 is constant; hence the space complexity of Binary Search is  $\theta(1)$ .



# Recursive Binary Search



The recursive approach of Binary search is similar to the iterative one. It assumes that every time a part of the array is selected for search, We can perform a Binary search on that array recursively.

## **ALGORITHM BinarySearch (A[ ], Low, High SearchKey)**

➤ **ALGORITHM BinarySearch (A[ ], Low, High SearchKey)**

➤ BEGIN:

➤ IF Low <= High THEN

➤ Mid =  $\lfloor (\text{Low} + \text{High}) / 2 \rfloor$

➤ IF A[Mid] == SearchKey THEN

➤ RETURN Mid

➤ ELSE

➤ IF SearchKey < A[Mid] THEN

➤ BinarySearch(A[ ], low, Mid-1, SearchKey)

➤ ELSE

➤ BinarySearch(A[ ], Mid+1, High, SearchKey)

➤ RETURN -1

➤ END;

The recursive Binary Search approach is easy to write, but it increases the space complexity because of pending activation records. The space taken by recursive Binary Search is  $O(\log_2 N)$

# Comparison of Linear Search with Binary Search

	Linear Search	Binary Search
Working	Linear search iterates through all the elements and compares them with the key which has to be searched.	Binary search wisely decreases the size of the array which has to be searched and compares the key with the mid element every time.
Prerequisites	Data can be random or sorted the algorithm remains the same, so there is no need for any pre-work.	It works only on a sorted array, so sorting an array is a prerequisite for this algorithm.
Use Case	We are generally preferred for smaller and randomly ordered datasets.	We are preferred for comparatively larger and sorted datasets.
Effectiveness	Less efficient in the case of larger datasets.	More efficient in the case of larger datasets.
Time Complexity	Best-case complexity - $\Omega(1)$ Worst-case complexity - $O(n)$	Best-case complexity - $\Omega(1)$ Worst-case complexity- $O(\log_2 n)$

# Ternary Search



## Ternary Search

Ternary Search is nothing, but it is a variation of Binary Search. The Binary Search divides the search half into two, and the Ternary divides into 3. In Ternary Search, if  $x=3$ , then divide the search intervals into 3 approximately equal subparts. There will be two middle values: middle1 and middle2.

Let us suppose that the element to be searched is Searchkey,

- 1- Compare the Searchkey with middle1 and middle2
- 2- If Searchkey is found, then return else decide in which search interval item belongs in next step.

Part1 Part2 Part3

Let us assume that array size is  $n$ ,  $beg=0$  and  $end=n-1$ .



## ALGORITHM TernarySearch(A[n], Beg, End, SearchKey)

**BEGIN:**

IF End < Beg THEN

RETURN -1

Middle1 = Beg + (End - Beg) / 3

Middle2 = Middle1 + (End - Beg) / 3

IF A[Middle1] == SearchKey THEN

RETURN Middle1

IF A[Middle2] == SearchKey THEN

RETURN Middle2

IF A[Middle1] > SearchKey THEN

RETURN TernarySearch(A, Beg, Middle1 - 1, SearchKey)

IF A[Middle2] > SearchKey THEN

RETURN TernarySearch(A, Middle1 + 1, Middle2 - 1, SearchKey)

RETURN TernarySearch(A, Middle2 + 1, End, SearchKey)

**END;**

# Analysis of Algorithm



## Time Complexity

The time complexity of ternary search =  $\log_3 n$  because total function call is  $\log_3 n$ , but in binary search complexity is  $\log_2 n$  i.e. number of function call is more than ternary search. It seems the ternary search is better but ternary search needs more comparison than binary search.

It means binary search takes less comparison than ternary search so overall complexity is not reduced significantly.

# Jump Search

## Jump Search

Jump search is also a variation of binary search in which we cannot compare with each element but jump to a fixed interval of size say  $x$  and reach that particular interval where data exist and apply binary search or linear search based on problem situation in that interval.

$x$	$2x...$	$kx$
-----	---------	------

Let us suppose that the element to be searched is item with elements at given interval boundaries  $a[x], a[2x], \dots, a[kx]$  and find  $z$  such that  
 $a[zx] < z \leq a[(z+1)x]$

Let us assume that array size be  $n$  and element to be find is item.

# Algorithm



**ALGORITHM JumpSearch(A[ ],n, SearchKey)**

**BEGIN:**

$s = \sqrt{n}$

$p = 0$

**WHILE**  $A[\min(s, n) - 1] < \text{item}$  **DO**

$p = s$

$s = s + \sqrt{n}$

**IF**  $p \geq n$

**RETURN** -1

**WHILE**  $A[p] < \text{SearchKey}$  **DO**

$p = p + 1$

**IF**  $p == \min(s, n)$  **THEN**

**RETURN** -1

**IF**  $A[p] == \text{SearchKey}$  **THEN**

**RETURN**  $p$

**RETURN** -1

**END;**



# Analysis



The most optimal interval size is  $\sqrt{n}$  because if the total number of elements is  $n$  and we divide it into  $k$  parts, then we have  $n/k$  intervals. Now apply Linear search in any interval, then it takes  $k-1$  comparison, so the equation becomes  $n/k + k-1$  for  $k = \sqrt{n}$  it gives optimize value.

**Complexity:** Time complexity of jump search is between of linear and binary search i.e.  $O(n)$  to  $O(\log n)$ .

## Exponential Search

Exponential search is also a variation of binary search. In this, first of all find range inside the given array and after that apply binary search or linear search in that range. It is also equivalent to jump search if interval of jump search is exponential (e.g. power of 2 or 3). Let us assume that array size be  $n$  and the element to be find is item.

**ALGORITHM** `jumpSearch(A[n], n, SearchKey)`

**BEGIN:**

    IF  $A[0] == \text{SearchKey}$  THEN  
        RETURN 0

$i = 1$

WHILE  $i < n \ \&\& \ A[i] < \text{item}$  DO

$i = i * 2$

IF  $A[i] == \text{SearchKey}$  THEN  
    RETURN  $i$

RETURN `BinarySearch(A, i/2, min(i, n), SearchKey)`

**END;**

# Analysis



- **Complexity:** Time complexity is  $O(\log i)$  best case and in the worst case it becomes  $O(\log n)$
- 
- **Problem:** If you have an incoming infinite stream of data in sorted order, how will you search a particular item in that stream.
- 
- **Solution:** Due to an infinite stream of data, we only know the start point of data but there is no idea about the endpoint. In this particular situation, exponential search is best suited.

# Interpolation Search



## Interpolation Search

Interpolation search is also a variation of binary search and performs on uniform distribution of data. In this search, do not move Middle positions but move low or high index depending on which is closer to searched item? This is just like binary search except for the calculation of the middle point.

$$\text{Middle} = \text{Beg} + \text{End} / 2$$

$$\text{Middle} = \text{Beg} + (\text{SearchKey} - \text{A}[\text{Beg}]) * (\text{End} - \text{Beg}) / (\text{A}[\text{End}] - \text{A}[\text{Beg}])$$

Let us assume that array size be n, Beg=0 and End=n-1



# Algorithm



**ALGORITHM** InterpolationSearch(A[n], Beg, End, SearchKey)

**BEGIN:**

IF End < Beg THEN

RETURN -1

Middle = Beg + (SearchKey - A[Beg]) \* (End - Beg) / (A[End] - A[Beg])

IF A[Middle] == SearchKey THEN

RETURN Middle

IF A[Middle] > SearchKey THEN

RETURN InterpolationSearch(A, Beg, Middle - 1, SearchKey)

RETURN InterpolationSearch(A, Middle + 1, End, SearchKey)

**END;**

# Analysis of Algorithm



## Time Complexity

Time complexity is  $O(\log_2 \log_2 n)$

## Problem

Find out the peak element in the given number of items that is not sorted. A peak element is that element that is greater than both its neighbors.

Let elements are 4, 6, 11, 8, 5. Here 11 is the peak element.

## ALGORITHM PeakElement(A[n], Beg, End,n)

### BEGIN:

$m = \text{Beg} + (\text{End} - \text{Beg}) / 2$

IF  $(m == 0) \parallel (A[m-1] \leq A[m]) \&\& (m = n-1) \parallel (A[m+1] \leq A[m])$  THEN

    RETURN A[m]

IF  $m > 0 \&\& A[m-1] > A[m]$  THEN

    RETURN PeakElement(A, Beg, m-1, n)

ELSE

    RETURN PeakElement(A, m+1, end, n)

### END;

# Index Sequential Search

## Index Sequential Search

- It is a searching technique that uses sequential searching and random-access searching methods. In this searching, a given sorted array of  $n$  elements is divided into groups based on the group size. Then we create an index array that contains the starting index of each group. This index array also stores the indexes of each group in increasing order.
- If we want to search an element called a key in the given array, we find the index group to present that search element.
- Following are the steps to implement Index Sequential Search
  - Suppose an array  $A$  of  $N$  elements is given in which elements are stored in sorted order. Now divide the array into groups according to the group size.
  - Now stores the starting index of each group into the index array. Here each element in the index array points to the group of elements. From this index array, we can also get the starting and end index of that group where the search element is expected to be present.
  - Read the search element called KEY.
  - Now compare the item with the first element; if the item is greater than the first element of that group, then moves to the next group; otherwise, perform the sequential search in the previous group. Repeat this step until all groups are processed.

# Application

## Application

This index sequential search or access to search or access records in the Database. It accesses database records very quickly if the index table is organized correctly. The main advantage of the indexed sequential is that it reduces the search time for a given item because sequential search is performed on the smaller range compared to the large table.

### Indexed Sequential Search

It is based on **sequential** and **random access** searching method.  
It search element according to groups



#### Steps:

1. Read the search element from the user.
2. Divide the array into groups according to group size.
3. Create index array that contains starting index of groups
4. If group is present **and** first element of that group is **less than or equal to** key element go to next group  
Else apply linear search in previous group
5. Repeat step 4 for all groups

0	1	2	3	4	5	6	7	8
12	15	17	21	28	36	57	81	99

A =

0	1	2

Index =

key = ; n = ; GS =



# Algorithm

**ALGORITHM: IndexSequentialSearch(A[ ], N, KEY)**

**BEGIN:**

Ind[ ], indele[ ]

n1=0, si, ei, flag = 0

FOR i = 0 TO N STEP+3 DO

Indele[n1] = A[ i ]

ind[n1] = i

n1 = n1+1

IF KEY<Indele[0] THEN

WRITE("ITEM NOT FOUND")

ELSE

FOR i = 1 TO n1+1 DO

IF KEY<Indele[i] THEN

si = ind[i-1]

ei = ind[i]

flag = 1

BREAK

IF flag = 0 THEN

si = ind[i-1]

ei = N-1

FOR i = si TO ei DO

IF KEY == A[i] THEN

j = 1

BREAK

IF j == 1 THEN

WRITE(i+1)

ELSE

WRITE("No. not found")

**END;**

- **Time Complexity:  $\Theta(N/K)$**
- If the index is created by selecting each  $k^{\text{th}}$  element in the list or the size of the group is  $K$  then the size of the index is  $N/K$ , upon which sequential search is executed. This complexity can be further reduced if we apply binary search on the index array to select the search array.
- In that case it will take  $\log_2(N/K)$  time to search in index array and order of  $K$  time to perform the sequential search.
- 
- **Space Complexity:  $\Theta(N/K)$**
- If index is created by selecting each  $k^{\text{th}}$  element in the list, size of index is  $N/K$ . This is the additional space than the original array. In case of Index array is given, space complexity will be  $\Theta(1)$ .

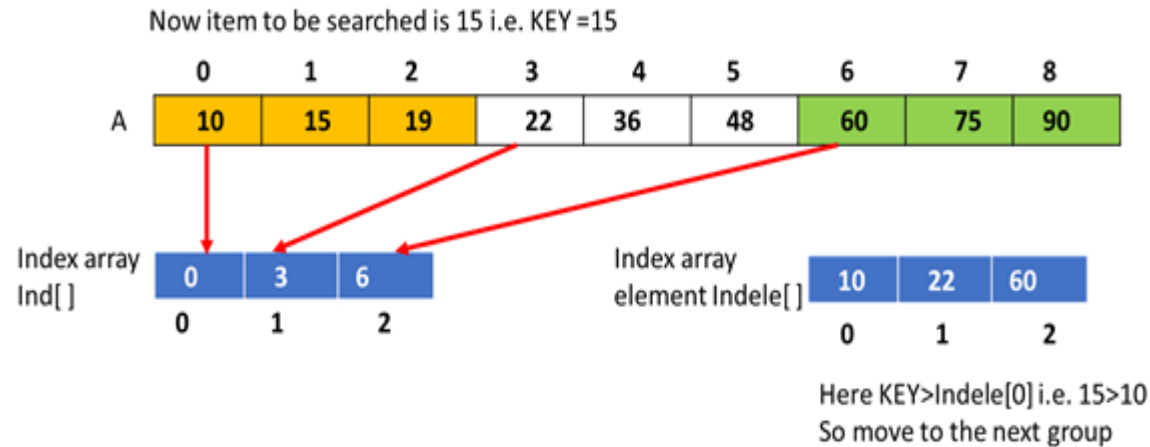
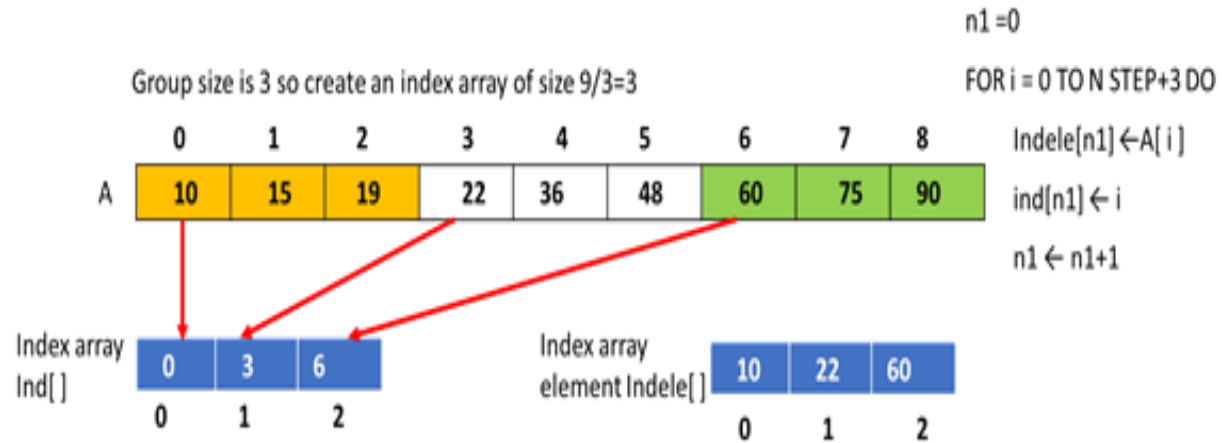
# Analysis

Example:

An Array of  $N=9$  elements is given in sorted order.

	0	1	2	3	4	5	6	7	8
A	10	15	19	22	36	48	60	75	90

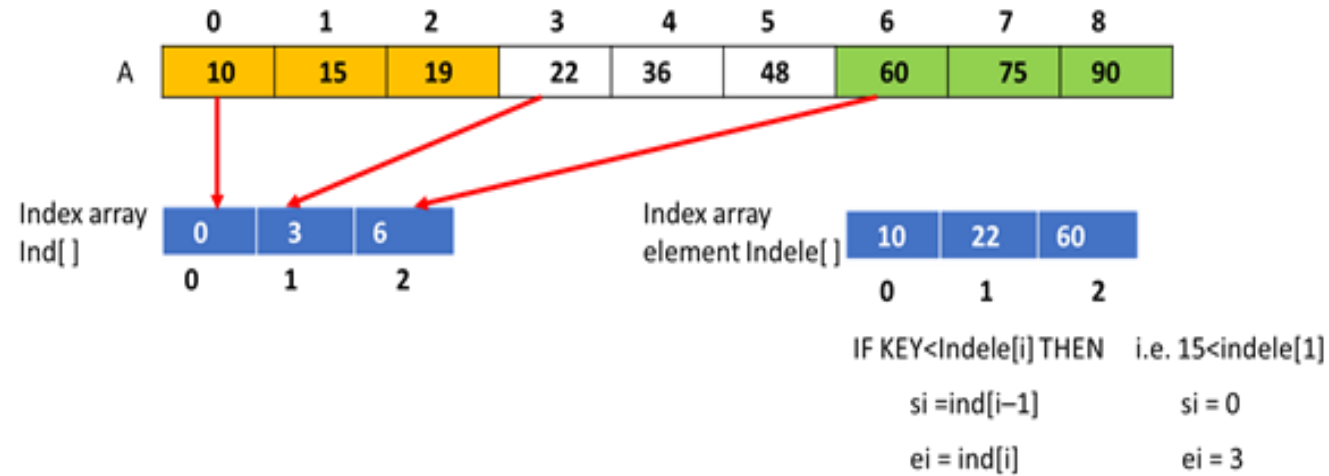
# Example





# Example

Now item to be searched is 15 i.e. KEY=15



Now perform the sequential search on the Group having starting index 0 and end index <3

	0	1	2	3	4	5	6	7	8
A	10	15	19	22	36	48	60	75	90

Here 15 <> 10  
move to next  
element

```
FOR i = si TO ei DO
  IF KEY == A[i] THEN
    j = 1
    BREAK
  IF j == 1 THEN
    WRITE(i+1)
  ELSE
    WRITE("No. not found")
```

# Example

Now perform the sequential search on the Group having starting index 0 and end index <3

	0	1	2	3	4	5	6	7	8
A	10	15	19	22	36	48	60	75	90

Here 15 == 10  
Item is found

```
FOR i = si TO ei DO
  IF KEY == A[i] THEN
    j = 1
    BREAK
IF j == 1 THEN
  WRITE(i+1)
ELSE
  WRITE("No. not found")
```



# HASHING



## Introduction

- 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.
  - 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.
- 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*.



# Hashing

- if there are four elements 3, 5, 7, 2. Then in order to search any of these, we can take array of size 7 and can search in  $O(1)$  constant time

0	1	2	3	4	5	6
0	1	2	3	4	5	6

The disadvantage of this method is the wastage of space. For example, suppose we have three elements 1, 23, 100. In this case, we need to take an array of size 100.

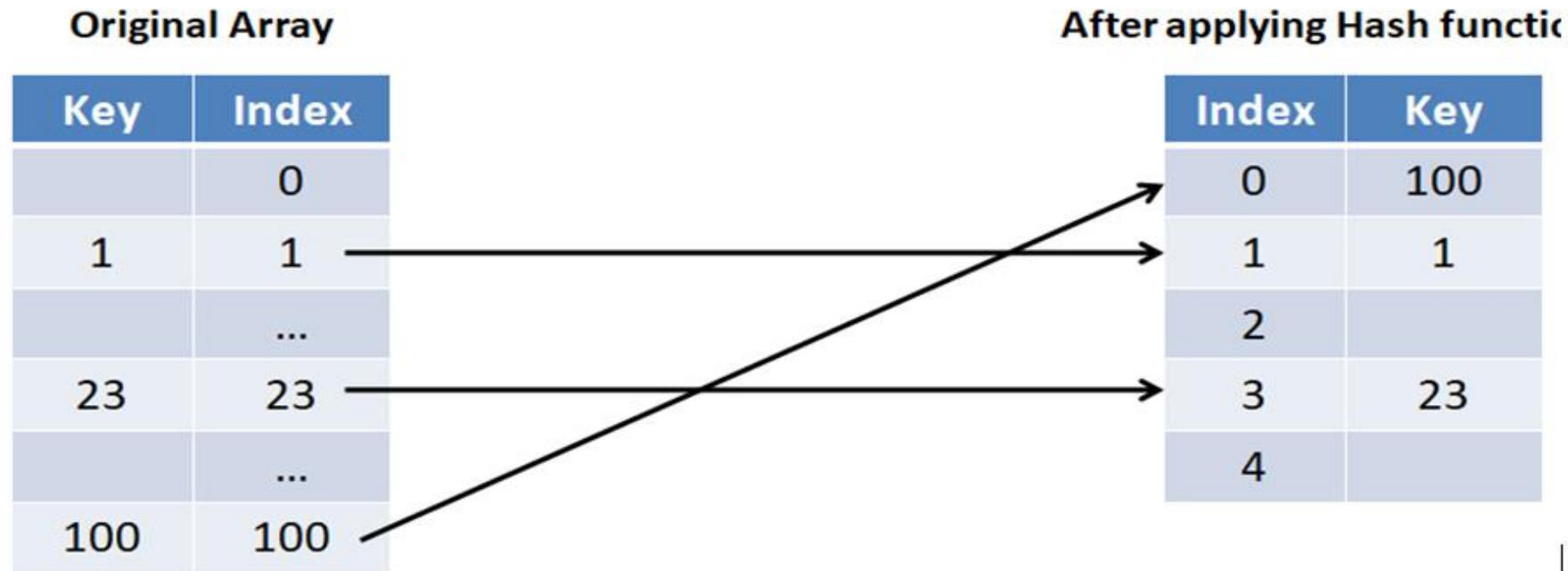
	1	...	23	...		100
0	1		23			100

# Hashing

- In *hashing*, large keys are *converted* into *small keys* by using *hash functions*.
- The values are then *stored in a data structure* called *hash table*.
- The *idea* of hashing is to *distribute entries* (key/value pairs) *uniformly across an array*.
- Each element is assigned a key (converted key).
- By using that key you can access the element in **O(1)** time. Using the key, the *algorithm* (hash function) computes an index that suggests where an entry can be *found* or *inserted*.

# Hashing

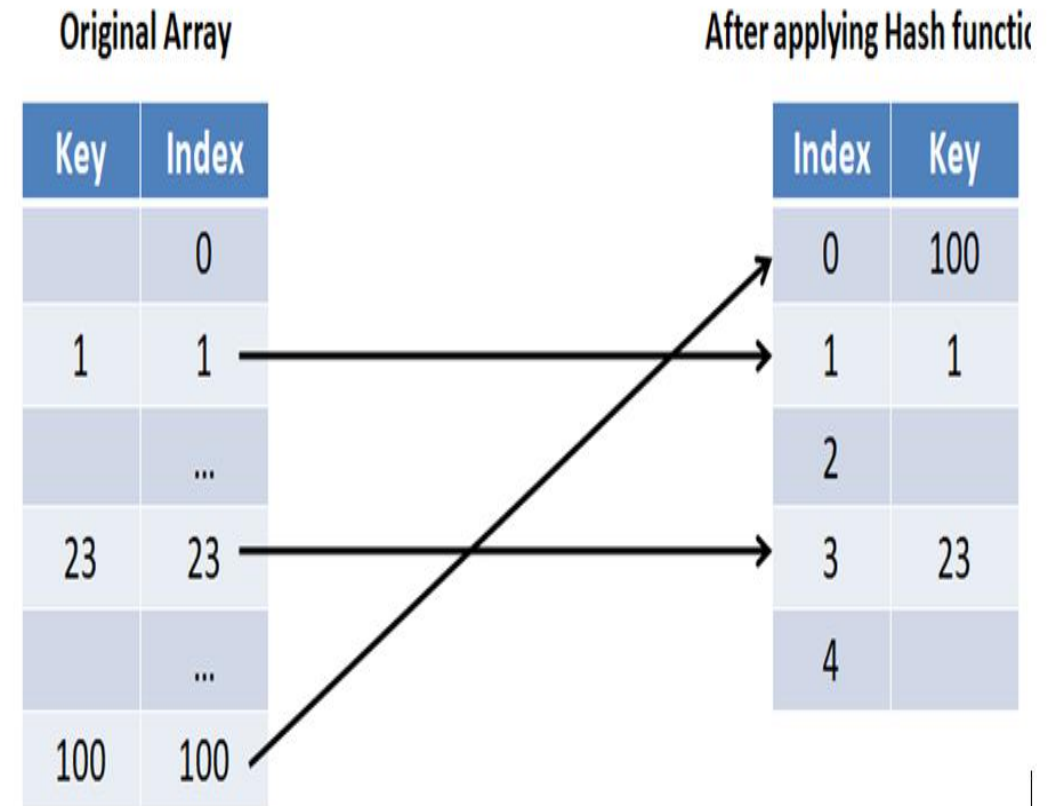
The above elements are mapped using hash function  $H(x) = H(x) \bmod 5$



Suppose we want to search a topic hashing in a Data Structure Book. Then, instead of using linear search or binary search technique, we can directly use the help of the index page and can see its exact page number and search this in  $O(1)$  time.

# Definition of Hashing

- The process of transforming an element into a secret element using a hash code is known as hashing.
- The mathematical function used for transforming an element into a mapped one or to secret code is known as a Hash Function.





# Hashing Functions

## Modulo Method (Division Modulo)

This method computes hash code using the division method. The simple formula for calculating hash code is given by

$$\text{Hash Code}(\text{key}) = \text{key} \bmod \text{HashTableSize}$$

The size of Hash Table is decided based on the input data set.

**Input elements** as 35, 44, 22, 19, 11, 20, 43, 6, 88, 27

$$\text{Hash code}(35) = 35 \bmod 10 = 5,$$

$$\text{Hash code}(44) = 44 \bmod 10 = 4$$

0	1	2	3	4	5	6	7	8	9
20	11	22	43	44	35	6	27	88	19



# Hashing Functions



Consider a Hash Table of Size 100,  
The Hash code for 123,223,323,423

$$123 \bmod 100 = 3$$

$$223 \bmod 100 = 3$$

$$323 \bmod 100 = 3$$

$$423 \bmod 100 = 3$$

We cannot store these keys at the same location in the Hash Table

**WHAT CAN BE DONE IN SUCH CASE??**

# Hashing Functions

We can divide the keys with some number with the least factors, a Prime number

**ALGORITHM NearestPrime(N)**

**BEGIN:**

FOR i = N TO 2 STEP -1 DO

    Prime = TRUE

    FOR j = 2 TO SQRT(N) AND Prime==TRUE DO

        IF i%j == 0 THEN

            Prime = FALSE

    IF Prime == TRUE THEN

        RETURN i

**END;**

**ALGORITHM DivisionHash (Key, Prime)**

**BEGIN:**

    RETURN key % Prime

**END;**

**Disadvantage:** This method may suffer from the collision. Two elements when converted to hash function, if result in having one hash code then collision is said to have occurred.

# Hashing Functions

## Mid Square Method

unique key is extracted from the middle of the square of the key

If the number of digits of the highest possible index in the chosen hash table (k), then this hashing process suggests picking k digits from the square of the keys to act as the hash code (if the hash table size is in the powers of 10) else modulus is taken of these mid k digits with the table size.

Key	104
Key <sup>2</sup>	10816
H(K)	10816

Key	4012
Key <sup>2</sup>	16096144
H(K)	16096144

# Hashing Functions

hash table having size N. Each hash table location has an address of k digits.

**ALGORITHM** MidsquareHash(key, k, N)

**BEGIN:**

$X = \text{key} * \text{key}$

$Y = X / 10^k$

$Z = Y \% 10^k$

$H = Z \% N$

RETURN H

**End;**

## Folding Method

- Divide the key into equal size of pieces (of the same length as that of the length of the largest address in the table size) and then these are added together.
- Modulus is taken of sum with the table size, which results in the hash code.

$$H(k) = \text{sum} \bmod N$$

Table size (N) be 1000, i.e. addresses will range from 0 – 999. The largest address is of 3 digit. If the key is 12345678, breaking it down into groups of 3 digits each.

$$12 + 345 + 678 = 1035$$

$$\text{Hash code} = 1035 \% 1000 = 35$$



# Hashing Functions

If the Table size (N) is 13, i.e. address will range from 0 – 12. Largest address is of 2 digits, the key will be divided into groups of 2 digits each. If the key is 12345678,

$$12+34+56+78 = 180$$

Hash Code =  $180 \% 13 = 11$

**Algorithm FoldingHash(key,k,N)**

**BEGIN:**

Sum=0

WHILE key!=0 DO

Sum = Sum + key %  $10^k$

key = key /  $10^k$

H=Sum%N

RETURN H

**END;**

# Hashing Functions

A variation of the folding method is Reverse Folding, in which either the odd group or the even group is reversed before addition.

## Algorithm ReverseFoldingHash(key,k,N)

**BEGIN:**

Sum=0

i=1

WHILE key!=0 DO

IF  $i \% 2 == 1$  THEN

Sum = Sum + key %  $10^k$

ELSE

Sum = Sum + Reverse(key %  $10^k$ )

i = i + 1

key = key /  $10^k$

H=Sum%N

RETURN H

**END;**

# Collision Resolution in Hashing



The situation when the location found for two keys are same, the situation can be termed as collision.

$$H:Key_1 \rightarrow L$$

$$H:Key_2 \rightarrow L$$

As two data values cannot be kept in the exact location, collision is not desirable situation.

Avoiding collisions completely is difficult, even with a good hash function. Some method should be used to resolve this.

# Collision Resolution in Hashing



There are two known methods of collision resolution in Hashing.

- Open Addressing
- Chaining

# Collision Resolution in Hashing



## Open Addressing

Every key considers the entire table as the storage space. Thus, if it does not find the appropriate place for storage through the hash function, it tries to find the next free available slot.

- There are 3 different Open addressing mechanisms named as
  - Linear Probing
  - Quadratic Probing
  - Rehashing/Double hashing



# Collision Resolution in Hashing



## Linear Probing

If the key cannot be stored/searched at the given hash location, try to find the next available free slot by traversing sequentially

If the table size is TS,

$$H(K,j) = (H(K) + j) \text{ modulus } TS$$

$$j=0, 1, 2, \dots$$

Sequence of investigation:

- $H(K)$
- $(H(K) + 1) \text{ modulus } TS$
- $(H(K) + 2) \text{ modulus } TS,$
- ...

Modulus is applied because the Hash table is considered to be circular in nature.

# Collision Resolution in Hashing

## Quadratic Probing

**Idea:** when there is a collision, check the next available position in the table using the quadratic formula:

$$H(K,j) = (H(K) + a*j + b*j^2) \text{ modulus TS}$$

$j = 0, 1, 2, \dots$

**if  $a=1/2, b=1/2$**

Sequence of investigation:

- $H(K)$
- $(H(K) + \frac{1}{2} + \frac{1}{2}) \text{ modulus TS}$
- $(H(K) + \frac{1}{2} * 2 + \frac{1}{2} * 2^2) \text{ modulus TS}$
- $(H(K) + \frac{1}{2} * 3 + \frac{1}{2} * 3^2) \text{ modulus TS and so on}$   
i.e.  $H(K), H(K)+1, H(K)+3, H(K)+6, \dots$

**if  $a=0, b=1$**

Sequence of investigation:

- $H(K)$
- $(H(K) + 1) \text{ modulus TS}$
- $(H(K) + 1 * 2^2) \text{ modulus TS}$
- $(H(K) + 1 * 3^2) \text{ modulus TS and so on}$   
i.e.  $H(K), H(K)+1, H(K)+4, H(K)+9, \dots$

# Collision Resolution in Hashing



## Rehashing/Double Hashing

The first hash function is used to find the hash table location and the second hash function to find the increment sequence

$$H(K,j) = (H(K) + j H'(K) ) \text{ modulus TS, } j=0,1,\dots$$

Sequence of investigation:

- $H(K)$
- $(H(K) + H'(K)) \text{ modulus TS}$
- $(H(K) + 2*H'(K)) \text{ modulus TS and so on}$
- $(H(K) + 3*H'(K)) \text{ modulus TS and so on}$

# Collision Resolution in Hashing

## Example

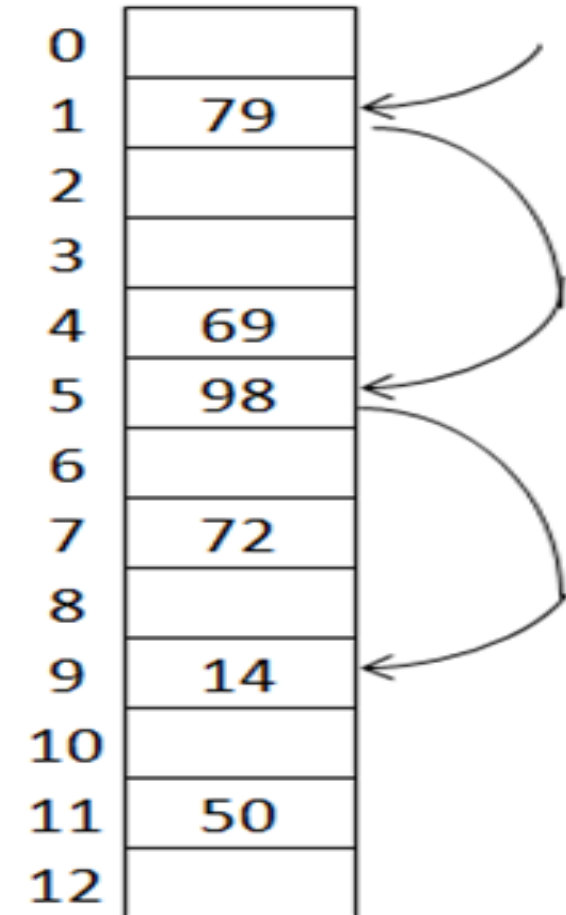
- $H(K) = k \text{ modulus } 13$
- $H'(K) = 1 + (k \text{ modulus } 11)$
- $H(K,i) = (H(K) + i H'(K)) \text{ mod } 13$

- **Insert key 14 in the Given Table**

$$H(14,0) = 14 \text{ modulus } 13 = 1$$

$$\begin{aligned} H(14,1) &= (H(14) + H'(14)) \text{ modulus } 13 \\ &= (1 + 4) \text{ modulus } 13 = 5 \end{aligned}$$

$$\begin{aligned} H(14,2) &= (H(14) + 2 H'(14)) \text{ modulus } 13 \\ &= (1 + 8) \text{ modulus } 13 = 9 \end{aligned}$$

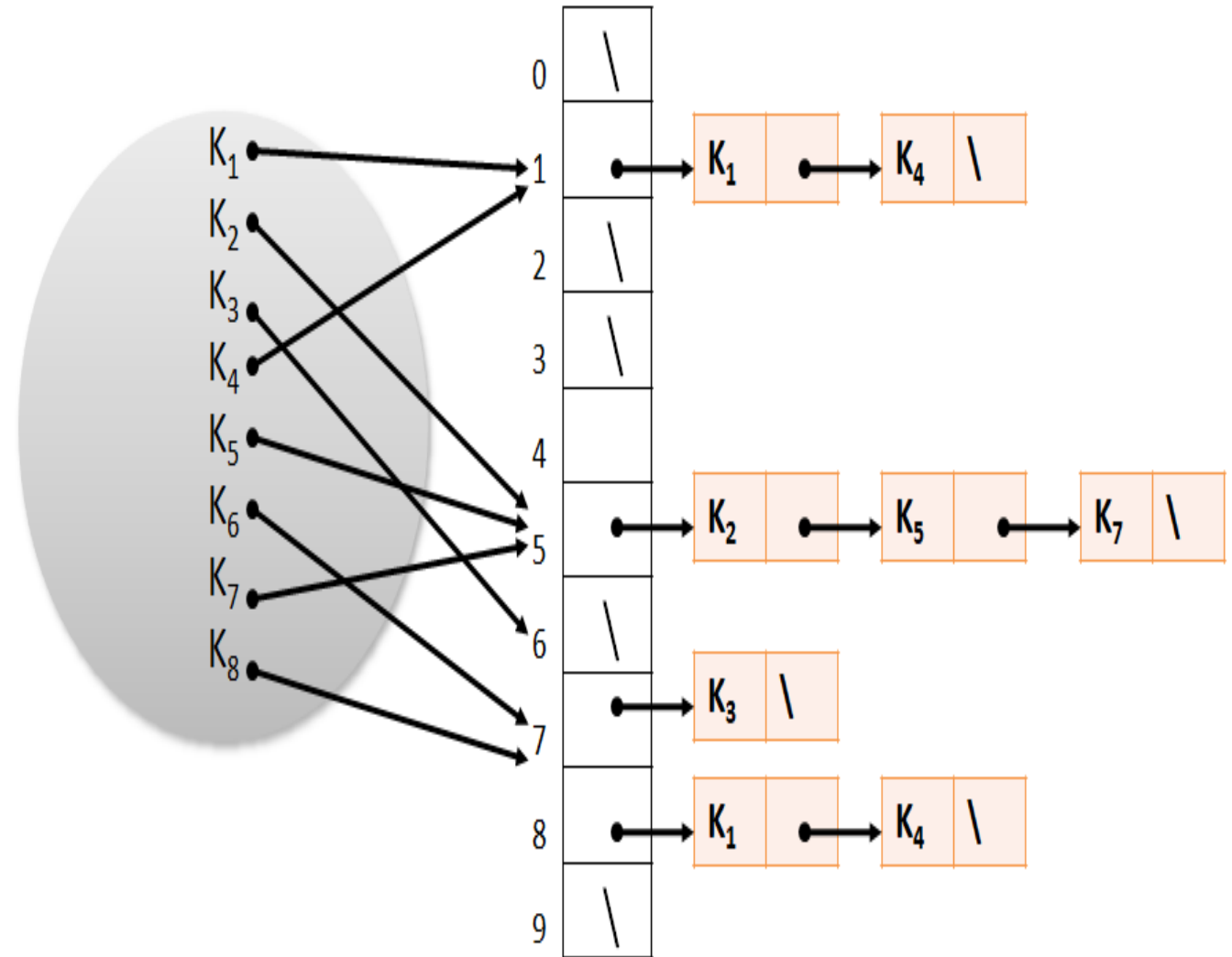


0	
1	79
2	
3	
4	69
5	98
6	
7	72
8	
9	14
10	
11	50
12	

# Collision Resolution in Hashing

## Chaining

- The chaining method takes the array of linked lists in contrast to the linear array for Hash Table.
- The keys with the same hash address go to the same linked list.





# Load Factor of a Hash Table



- Load factor is defined as

$$\lambda = N/TS$$

- N refers to the Total number of keys stored in the hash table
- TS refers to the Hash Table Size

# Summary



downloaded from pickywallpapers.com

Thank You

---