SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Searching and Hashing

Swati Mali
swatimali@somaiya.edu

Somaiya
TRUST

Outline

- Search concept
- Searching applications
- Linear Search
- Binary Search
- Hashed List Search
- Comparison of searching Techniques

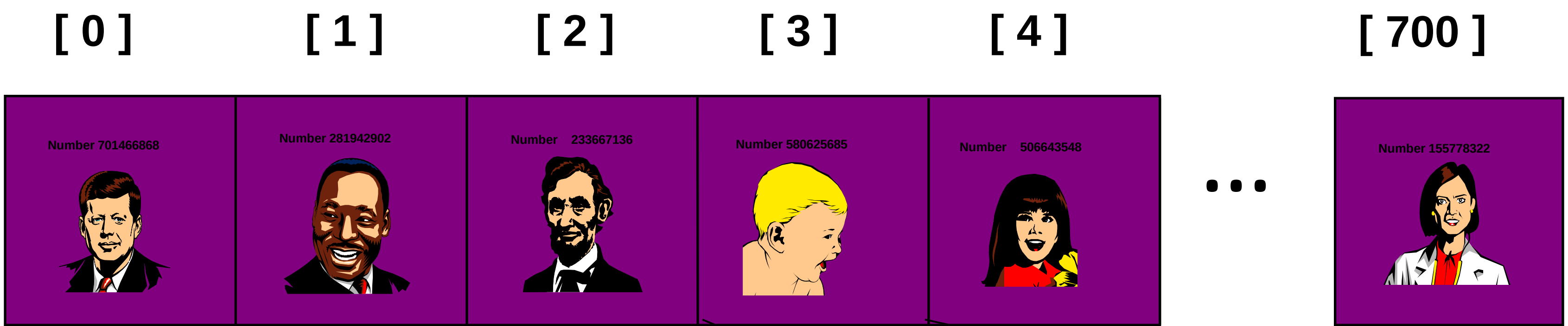
Searching

- Search is a process to retrieve information stored within some data structure, or calculated in the search space of a problem domain, either with discrete or continuous values.

Problem: Search

- We are given a list of records.
- Each record has an associated key.
- Give efficient algorithm for searching for a record containing a particular key.
- Efficiency is quantified in terms of average time analysis (number of comparisons) to retrieve an item.

Search



Each record in list has an associated key.
In this example, the keys are ID numbers.

Given a particular key, how can we efficiently retrieve the record from the list?

Searching applications

- Problems in combinatorial optimization, such as:
  - The vehicle routing problem, a form of shortest path problem
  - The knapsack problem: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
  - The nurse scheduling problem
- Problems in constraint satisfaction, such as:
  - The map coloring problem
  - Filling in a sudoku or crossword puzzle
- In game playing and especially multiplayer game playing, choosing the best move to make next (such as with the minmax algorithm)

Ref: https://en.wikipedia.org/wiki/Search_algorithm#Applications_of_Search_Algorithms

Searching applications

- Finding a combination or password from the whole set of possibilities
- Factoring an integer (an important problem in cryptography)
- User authentication – id search and password match
- Optimizing an industrial process, such as a chemical reaction, by changing the parameters of the process (like temperature, pressure, and pH)
- Retrieving a record from a database
- Finding the maximum or minimum value in a list or array
- Checking to see if a given value is present in a set of values

Linear Search

The list or data structure is traversed sequentially and every element is checked for presence of some information

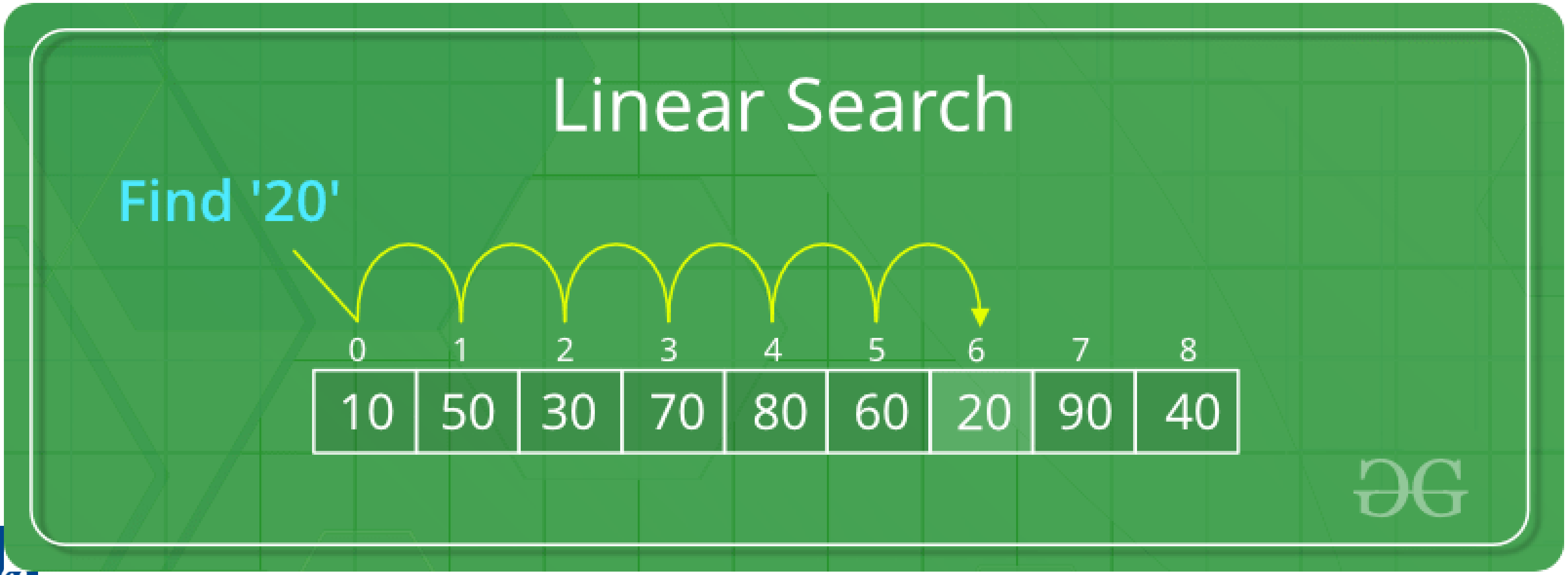


Image courtesy: https://www.geeksforgeeks.org/searching-algorithms/

Linear search Search

- Step through array of records, one at a time.
- Look for record with matching key.
- Search stops when
  - record with matching key is found
  - or when search has examined all records without success.

Pseudocode for Linear Search

```
// Search for a desired item in the n array elements
// starting at a[first].
// Returns pointer to desired record if found.
// Otherwise, return NULL
…
for(i = first; i < n; ++i )
        if(a[first+i] is desired item)
                return &a[first+i];

// if we drop through loop, then desired item was not found
return NULL;
```

Linear Search

- Advantages
  - **Fast searches of small to medium lists.**
  - **The list does not need to sorted.**
  - **Not affected by insertions and deletions**
- Disadvantages
  - **Slow searching of large lists**
  - **time taken to search the elements is proportional to the number of elements**.

Binary Search

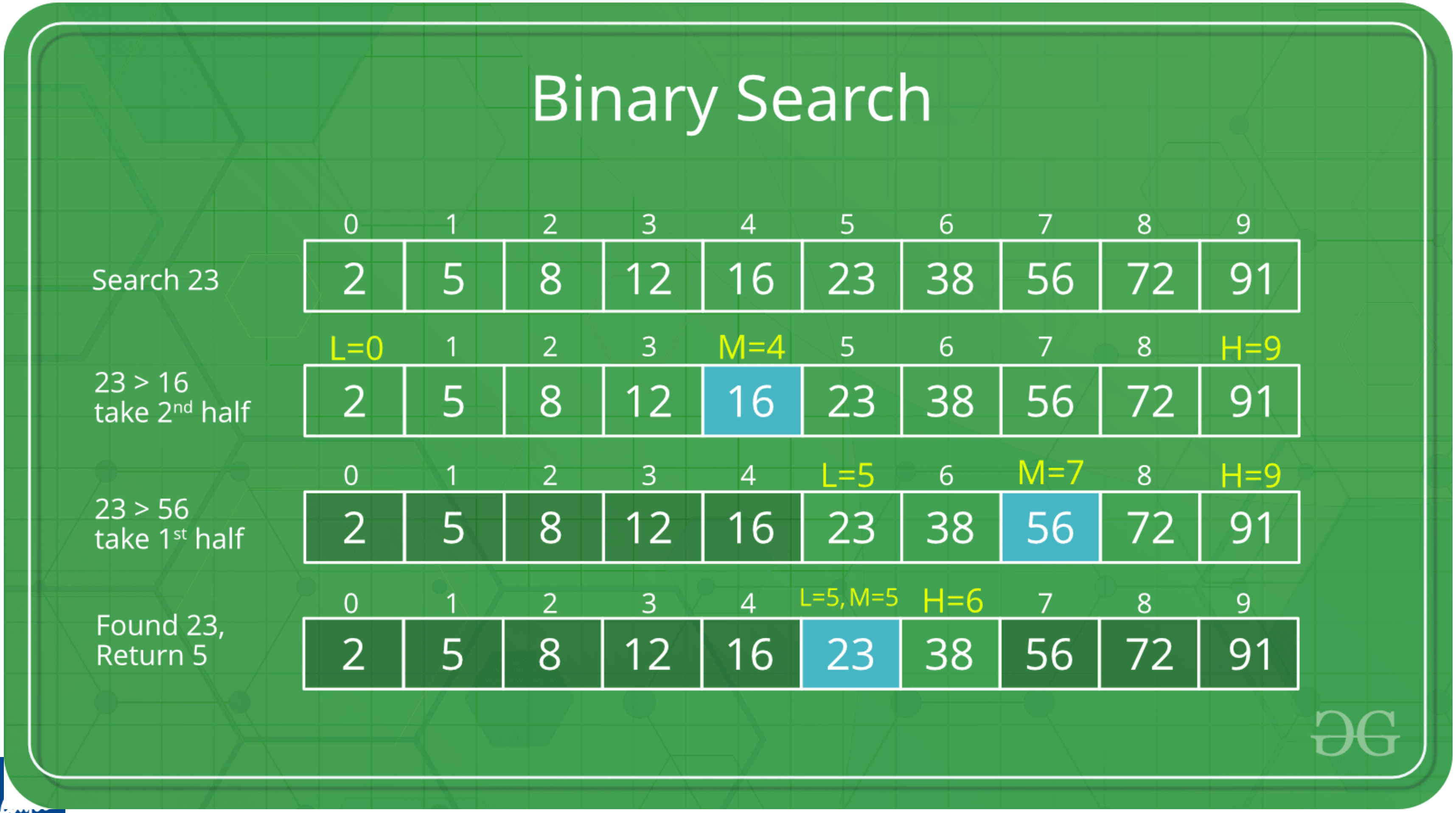
- Divide and conquer
- Needs the input to be sorted
- One of the fastest searching algorithms.

Binary Search algorithm

Algorithm integer binarySearch(int arr[], int low, int high, int SearchKey)
// Arr[0:n-1] array of N elements, SearchKey is key to be searched. Low and High are lowest and highest indices in array arr[0:n-1]

```
{
  if (low <= high)
  {
      mid = (low+ high)/2;
      if (arr[mid] == SearchKey)  return mid;
      if (arr[mid] > SearchKey)
         return binarySearch(arr, low, mid-1, SearchKey);
       else
         return binarySearch(arr, mid+1, high, SearchKey);
  }
   // if the element is not present in array
  return -1;
}
```

Binary search example



Binary search and number of comparisons

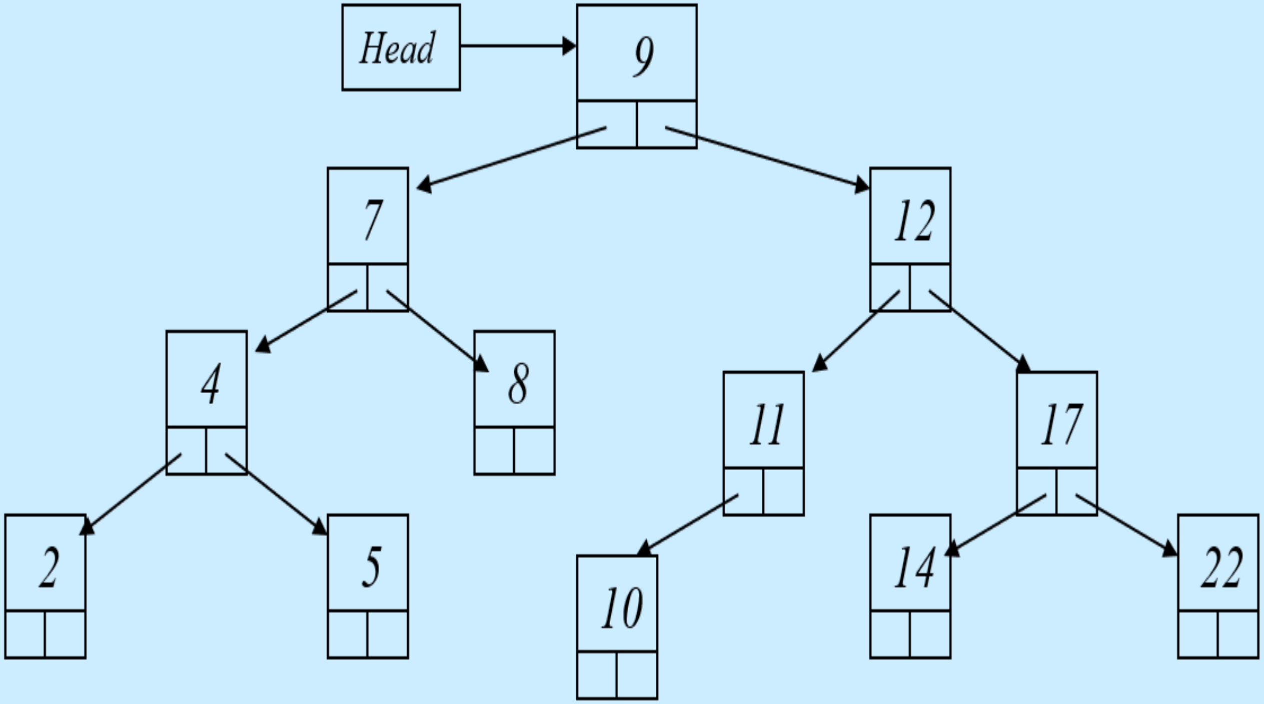
COMPARISON	LIST SIZE	"MIDDLE"	REMAINING ITEMS
1	10,000	5,001	5,000
2	5,000	2,501	2,500
3	2,500	1,276	1,275
4	1,275	638	637
5	637	319	318
6	318	160	159
7	159	80	79
8	79	40	39
9	39	20	19
10	19	10	9
11	9	5	4
12	4	3	2
13	2	2	1
14	1	1	0

Image courtesy: http://watson.latech.edu/book/algorithms/algorithmsSearching2.html

Binary Search

- ***The following is the binary search tree for the sorted set***

  { 2, 4, 5, 7, 8, 9, 10, 11, 12, 14, 17, 22}



Binary search

- Advantages
  - Efficient search
  - It indicates whether the element being searched is present before or after the current position in the list.
  - This information is used to narrow the search.
  - Works best for larger datasets
- Disadvantages
  - Recursion takes more space
  - Input needs to be sorted
  - Gets affected by insertions and deletions in the input list

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Hashing and hash tables

Somaiya TRUST

Concept of Hashing

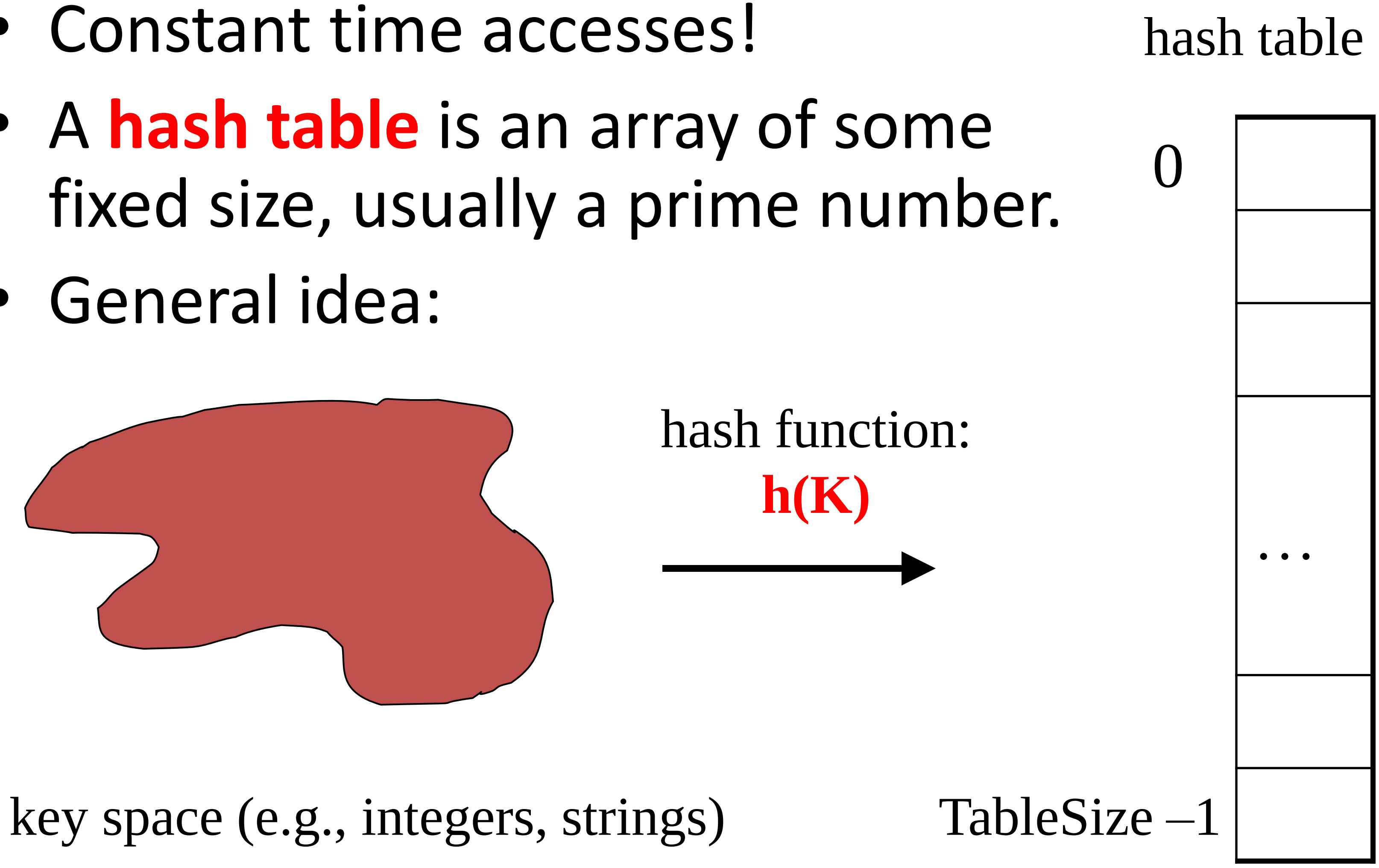
- In CS, a **hash table**, or a **hash map**, is a data structure that associates keys (names) with values (attributes).
  - Look-Up Table
  - Dictionary
  - Cache
  - Extended Array

Just An Idea

- Hash table :
  - Collection of pairs,
  - Lookup function (Hash function)
- Hash tables are often used to implement associative arrays,
  - Worst-case time for Get, Insert, and Delete is O(size).
  - Expected time is O(1).

Hash Tables

- Constant time accesses!
- A **hash table** is an array of some fixed size, usually a prime number.
- General idea:



SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Example

- key space = integers
- TableSize = 10

- **h**(K) = K mod 10

- **Insert**: 7, 18, 41, 94

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Somaiya TRUST

Hash Functions

1. **simple/fast** to compute,
2. Avoid **collisions**
3. have keys distributed **evenly** among cells.

Perfect Hash function:

Search vs. Hashing

- Search tree methods: key comparisons
  – Time complexity: O(size) or O(log n)
- Hashing methods: hash functions
  – Expected time: O(1)
- Is hashing better than searching?

Sample Hash Functions:

- key space = strings
- $s = s_0\ s_1\ s_2 \ldots s_{k-1}$

1. $h(s) = s_0 \bmod TableSize$
2. $h(s) = \left(\sum_{i=0}^{k-1} s_i\right) \bmod TableSize$
3. $h(s) = \left(\sum_{i=0}^{k-1} s_i \cdot 37^i\right) \bmod TableSize$

Some hash functions

- Middle of square
  - H(x):= return middle digits of x^2
- Division
  - H(x):= return x % k
- Multiplicative:
  - H(x):= return the first few digits of the fractional part of x*k, where k is a fraction.
    - advocated by D. Knuth in TAOCP vol. III.

Some hash functions II

- Folding:
  - Partition the identifier x into several parts, and add the parts together to obtain the hash address
  - e.g. x=12320324111220; partition x into 123,203,241,112,20; then return the address 123+203+241+112+20=699
  - Shift folding vs. folding at the boundaries
- Digit analysis:
  - If all the keys have been known in advance, then we could delete the digits of keys having the most skewed distributions, and use the rest digits as hash address.

Properties of a good hash function

- Randomization
- Less collisions

Choice of Hash Function

- Requirements
  - easy to compute
  - minimal number of collisions
- If a hashing function groups key values together, this is called clustering of the keys.
- A good hashing function distributes the key values uniformly throughout the range.

Collision Resolution

Collision: when two keys map to the same location in the hash table.

Two ways to resolve collisions:

1. Separate Chaining
2. Open Addressing (linear probing, quadratic probing, double hashing)

Open Addressing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Insert:
38
19
8
109
10

- **Linear Probing**: after checking spot h(k), try spot h(k)+1, if that is full, try h(k)+2, then h(k)+3, etc.

Linear Probing

$$f(i) = i$$

- Probe sequence:

  0^{th} probe = $h(k) \bmod TableSize$

  1^{th} probe = $(h(k) + 1) \bmod TableSize$

  2^{th} probe = $(h(k) + 2) \bmod TableSize$

  . . .

  i^{th} probe = $(h(k) + i) \bmod TableSize$

Quadratic Probing

Less likely to encounter Primary Clustering

$$f(i) = i^2$$

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + 1) mod TableSize

  2^{th} probe = (h(k) + 4) mod TableSize

  3^{th} probe = (h(k) + 9) mod TableSize

  . . .

  i^{th} probe = $(h(k) + i^2)$ mod TableSize

Quadratic Probing

- Assume a hash table with 11 slots (0 to 10), and we want to insert a key with the hash value of 4.
- Initially, we calculate the initial slot using the hash function:
  - Hashed Key: 4
  - Initial Slot: 4
- check if the initial slot (slot 4) is available. If it's empty, we can place the key there.
- However, let's assume that slot 4 is already occupied. In this case, we apply quadratic probing to find the next available slot.
  - First Quadratic Probe:
    - New Slot: 4 + (1^2) = 5
    - We check slot 5. If it's empty, we insert the key there. If not, we continue probing.
  - Second Quadratic Probe:
    - New Slot: 4 + (2^2) = 8
    - We check slot 8. If it's empty, we insert the key there. If not, we continue probing.
  - Third Quadratic Probe:
    - New Slot: 4 + (3^2) = 13
    - To keep the slot within the bounds of the hash table, we use modulo 11 (the number of slots):
      - New Slot: 13 % 11 = 2
    - We check slot 2. If it's empty, we insert the key there. If not, we continue probing.
  - Continue this process until an empty slot is found.

Quadratic Probing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Insert:
89
18
49
58
79

Double Hashing

$f(i) = i * g(k)$

where g is a second hash function

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + g(k)) mod TableSize

  2^{th} probe = (h(k) + 2*g(k)) mod TableSize

  3^{th} probe = (h(k) + 3*g(k)) mod TableSize

  . . .

  i^{th} probe = (h(k) + i*g(k)) mod TableSize

Double Hashing Example

h(k) = k mod 7 and g(k) = 5 – (k mod 5)

	76	93	40	47	10	55
0						
1				47	47	47
2		93	93	93	93	93
3					10	10
4						55
5			40	40	40	40
6	76	76	76	76	76	76
es	1	1	1	2	1	2

Linear Probing – example

- Divisor = b (number of buckets) = 17.
- Home bucket = key % 17.
- Collision resolution: [h(k)+1] mod bucketsize

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

- Insert pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

#collisions?

Quadratic Probing – example

- Divisor = b (number of buckets) = 17.
- Home bucket = key % 17.
- Collision resolution= $[h(k)+i^2)$ mod bucketsize

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- Insert pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

#collisions?

Separate chaining– example

- Divisor = b (number of buckets) = 17.
- Home bucket = key % 17.
- Collision resolution= separate chaining

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- Insert pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

Double hashing and rehashing

- **Double Hashing :**
  - Applying two functions at a time like : (H1(x) **operation** H2(x)) on any key item x ;
  - H1(x) and H2(x) are two different hash functions and **operation** can be as per necessity like multiplication/division/custom hash function
- **ReHashing :**
  - Applying Hashing function again and again on item in order to generate unique mapping value.

Thank you