HASHING

HASHING

- Two search algorithms- linear search and binary search.
- Running time for linear search: O(n)
- Running time for binary search: O(log n)
- Is there any way in which searching can be done in constant time i.e. O(1), irrespective of the number of elements in the array?
- There are two solutions to this problem.

HASHING

- To analyze the first solution let us take an example. In a small company of 100 employees, each employee is assigned an Emp_ID number in the range 0–99. To store the employee's records in an array, each employee's Emp_ID number act as an index into the array where this employee's record will be stored as shown in figure.

KEY	ARRAY OF EMPLOYEE'S RECORD
Key 0 [0]	Record of employee having Emp_ID 0
Key 1 [1]	Record of employee having Emp_ID 1
Key 2 [2]	Record of employee having Emp_ID 2
.....	.....
.....	.....
Key 98 [98]	Record of employee having Emp_ID 98
Key 99 [99]	Record of employee having Emp_ID 99

HASHING

- So now we can access the record of any employee, once we know the his EMP_ID. But this implementation is practically not feasible.
- Consider that same company use a five digit Emp_ID number as the primary key. In this case, key values will range from 00000 to 99999. we will need an array of size 1,00,000 of which only 100 elements will be used.

KEY		ARRAY OF EMPLOYEE'S RECORD
Key 00000	[0]	Record of employee having Emp_ID 00000
.....		.....
Key n	[n]	Record of employee having Emp_ID n
.....		.....
Key 99998	[99998]	Record of employee having Emp_ID 99998
Key 99999	[99999]	Record of employee having Emp_ID 99999

- Wasting of memory occurs as there are only 100 employees in the company.

HASHING

- Another good solution is to use just the last two digits of key to identify each employee.
- For example, the record of an employee with Emp_ID number 79439 will be stored in the array at index 39. Similarly, employee with Emp_ID 12345 will have its record stored in the array at the 45^{th} location.
- So, in the second solution, the elements are not stored according to the value of the key. Hence we need a way to convert a five-digit key number to two-digit array index. We need some function that will do the transformation.
- In this case, we will use the term **Hash Table** for an array and the function that will carry out the transformation will be called a **Hash Function**.

HASH TABLE

- **Hash Table** is a data structure in which keys are mapped to array positions by a hash function.
- A value stored in the Hash Table can be searched in O(1) time using a hash function to generate an address from the key.
- In a hash table, an element with key k is stored at index h(k) not k.
- This means, a hash function h is used to calculate the index at which the element with key k will be stored. Thus, the process of mapping keys to appropriate locations (or indexes) in a hash table is called **hashing**.

HASH FUNCTION

- Hash Function, h is simply a mathematical formula which when applied to the key, produces an integer which can be used as an index in the hash table to store element with key.
- The main aim of a hash function is that elements should be distributed randomly and uniformly.
- Hash function produces a unique set of integers within some suitable range. Such function produces no collisions. But practically, there is no hash function that eliminates collision completely.

HASH FUNCTION

Division Method:

- Simple method of hashing.
- The method divides x (key) by M (number of slots in the table) and then use the remainder thus obtained. In this case, the hash function can be given as

  $$h(x) = x \bmod M$$

- The division method is quite good for just about any value of M and since it requires only a single division operation, the method works very fast. However, extra care should be taken to select a suitable value for M.
- For example, M is an even number, then h(x) is even if x is even; and h(x) is odd if x is odd.

HASH FUNCTION

Division Method:

- If even keys are more likely than odd keys, then the division method will not spread the hashed values uniformly.
- In this case, the choice of hash function and table size needs to be carefully considered. It is best to choose M to be a prime number.
- **Example:** Calculate hash values of keys 1234 and 5462.

  Setting m = 97, hash values can be calculated as

  Solution: h(**1234**) = 1234 % 97 = **70**

  h(**5642**) = 5642 % 97 = **16**

HASH FUNCTION

Division Method:

- A potential drawback of the division method is that while using this method, consecutive keys map to consecutive hash values.
- On one hand, this is good as it ensures that consecutive keys do not collide, but on the other, it also means that consecutive array locations will be occupied. This may lead to degradation in performance.

HASH FUNCTION

Mid-Square Method:

The mid-square method works in two steps:

Step 1: Square the value of the key. That is, find k*k (i.e. k^2).

Step 2: Extract the middle r digits of the result obtained in Step 1.

- Algorithm works well because all the digits in the original key value contribute to produce the middle digits of the squared value.
- In the mid-square method, the same r digits must be chosen from all the keys. Therefore, the hash function can be given as:
- h(k) = s where s is obtained by selecting r digits from k^2.

HASH FUNCTION

Mid-Square Method:

Example: Calculate the hash value for keys 1234 and 5642 using the mid-square method. The hash table has 100 memory locations.

Solution: Note that the hash table has 100 memory locations whose indices vary from 0 to 99. This means that only two digits are needed to map the key to a location in the hash table, so r = 2.

When k = 1234, k^2 = 1522756, h (**1234**) = **27**

When k = 5642, k^2 = 31832164, h (**5642**) = **21**

Observe that the 3^{rd} and 4^{th} digits starting from the right are chosen.

HASH FUNCTION

Folding Method: The folding method works in the following two steps:

Step 1: Divide the key value into a number of parts. That is, divide k into parts k1, k2, ..., kn, where each part has the same number of digits except the last part which may have lesser digits than the other parts.

Step 2: Add the individual parts. That is, obtain the sum of k1 + k2 + ... + kn. The hash value is produced by ignoring the last carry, if any.

- **Note:** Number of digits in each part of the key will vary depending upon the size of the hash table.
- For example, if the hash table has a size of 1000, then there are 1000 locations in the hash table. To address these 1000 locations, we need at least three digits; therefore, each part of the key must have three digits except the last part which may have lesser digits.

HASH FUNCTION

Folding Method:

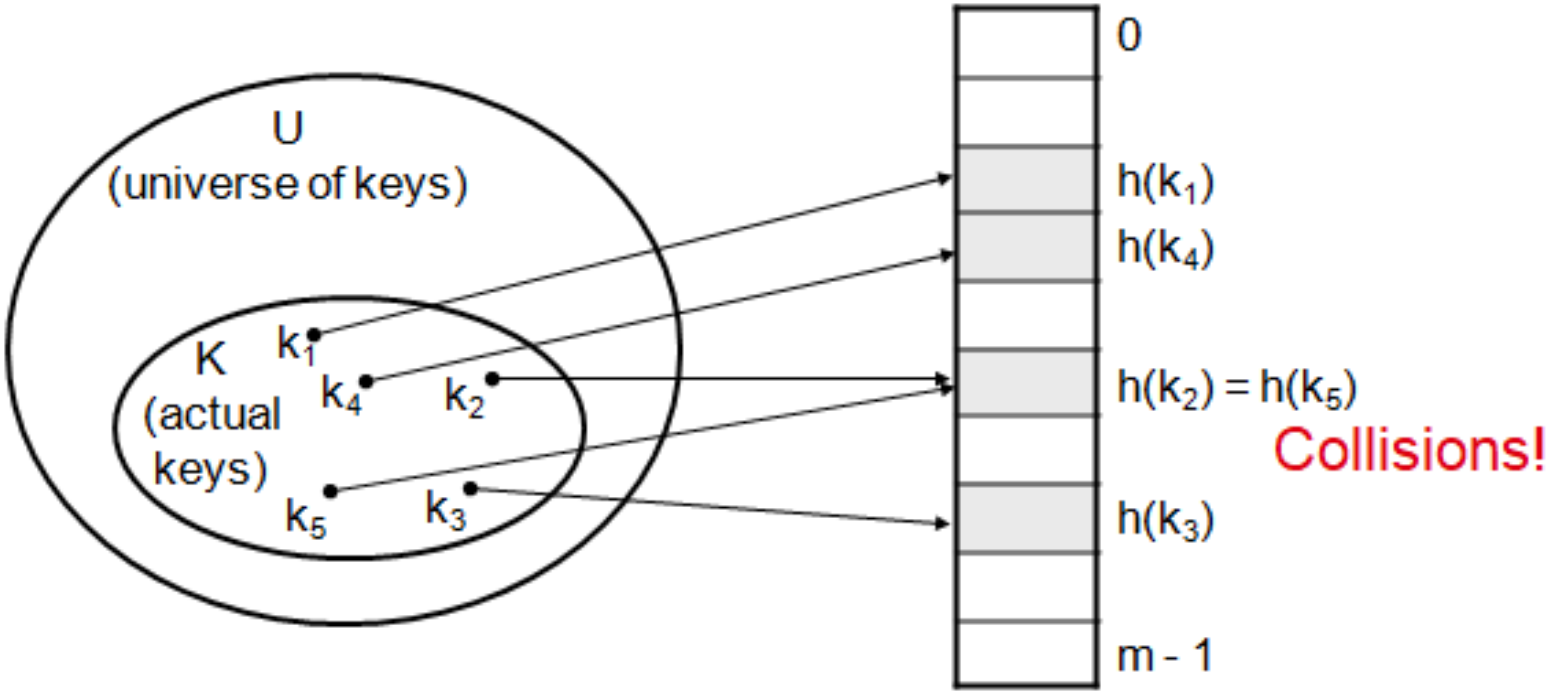
Example: Given a hash table of 100 locations, calculate the hash value using folding method for keys 5678, 321, and 34567.

Solution: Since there are 100 memory locations to address, we will break the key into parts where each part (except the last) will contain two digits. The hash values can be obtained as shown below:

Key	5678	321	34567
Parts	56 and 78	32 and 1	34, 56 and 7
Sum	134	33	97
Hash Value	34 (ignore the last carry)	33	97

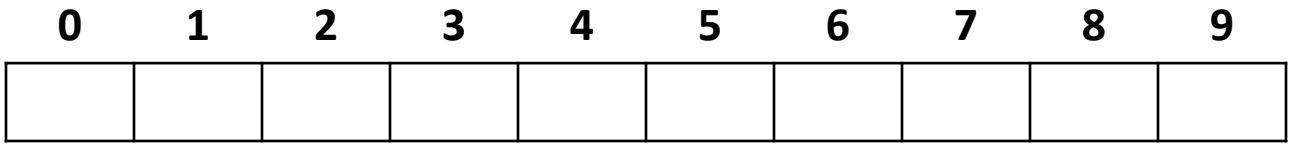
COLLISION

- Hash function maps two or more keys to the same slot or location!!



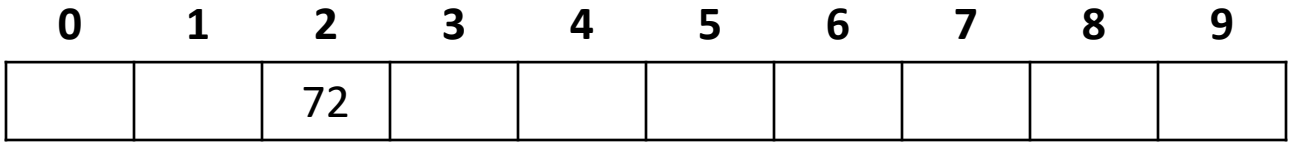
COLLISION

- **Example:** Consider a hash table of size 10. Insert the keys 72, 81, 92, and 101 into the table.



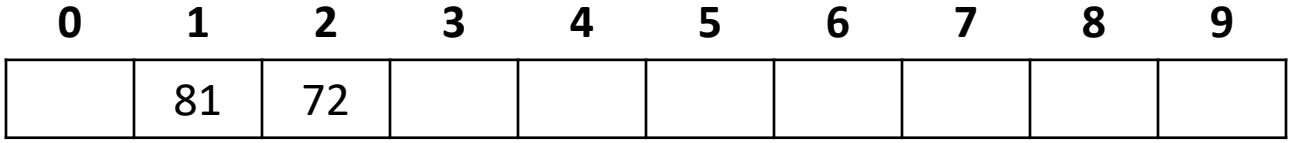
- Using Division method: Let h(k) = k mod m, Here m = 10
- Key = 72: h(72) = 72 mod 10 = 2

  Since T[2] is vacant, insert key 72 at this location.



- Key = 81: h(81) = 81 mod 10 = 1

  Since T[a] is vacant, insert key 81 at this location.



COLLISION

0	1	2	3	4	5	6	7	8	9
	81	72							

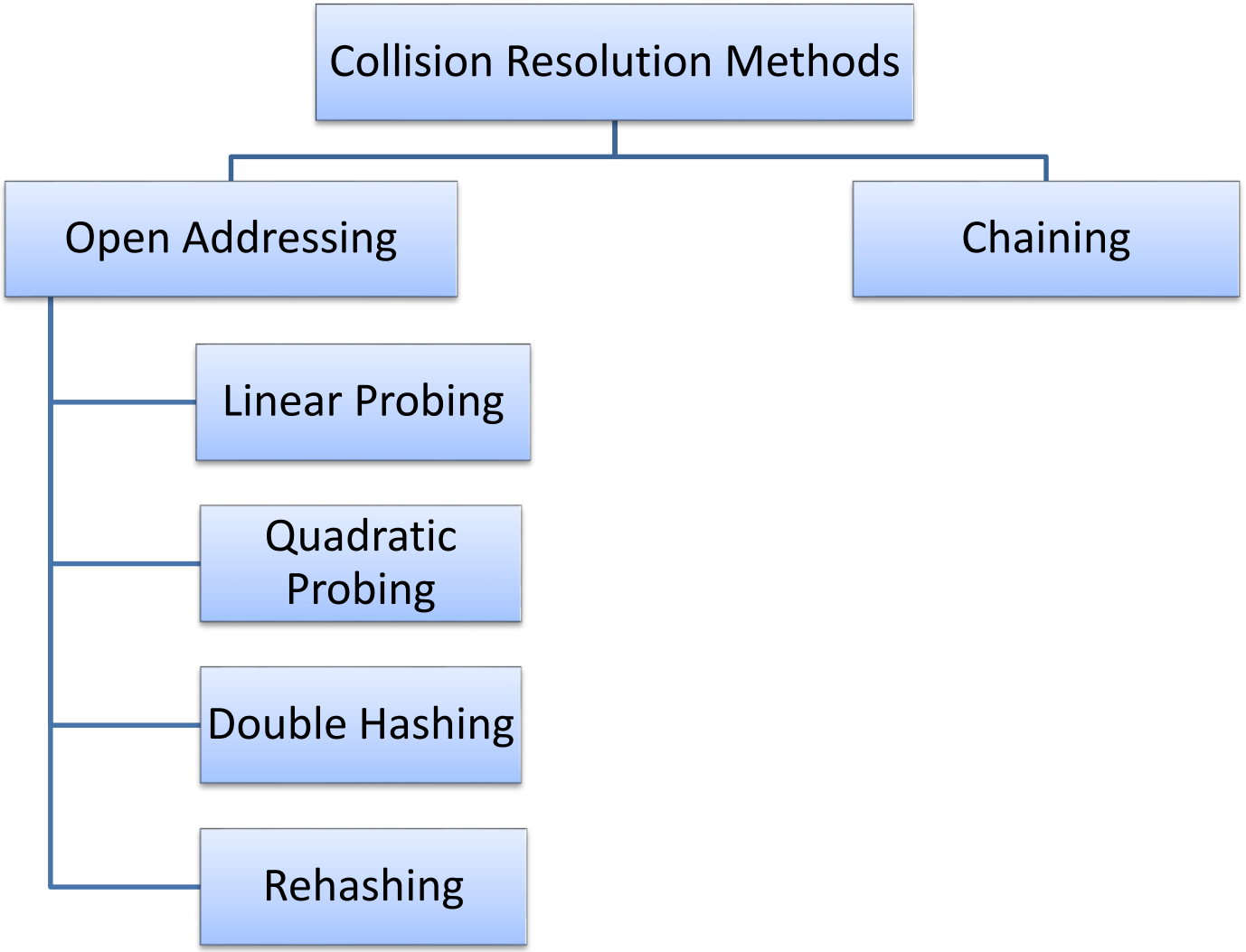
- Key = 92: h(92) = 92 mod 10 = 2

  Now T[2] is occupied, so we cannot store the key 92 in T[2]. **Obviously, two records can not be stored in the same location.**
- Key = 101: h(101) = 101 mod 10 = 1

  Now T[1] is occupied, so we cannot store the key 101 in T[1].

- **This is called collision.**
- **Method to solve collisions are called Collision Resolution Technique.**

COLLISION RESOLUTION TECHNIQUE



COLLISION RESOLUTION TECHNIQUE

Open Addressing Technique:

- In open addressing or closed hashing, we compute new location using a probe sequence and the record is stored in that location.
- The process of examining memory locations in the hash table is called probing.
- If even a single free location is not found, then we have an OVERFLOW condition.
- The hash table contains two types of values:
  1. sentinel values (e.g., –1)
  2. data values
- The presence of a sentinel value indicates that the location contains no data value at present.

LINEAR PROBING

- Simplest approach
- If the current location is used, try the next table location.
- Lookups walk along table until an empty slot is found.
- If a value is already stored at a location generated by h(k), then the following hash function is used to resolve the collision:
- h(k,0) = [h'(k) + i] mod m

  h'(k) = k mod m

  m is the size of the hash table

  i is the probe number that varies from 0 to m–1
- Probe sequence:
  - ✓ First location probed: h(k, 0) = [h'(k) + 0] mod m
  - ✓ Second location probed: h(k, 1) = [h'(k) + 1] mod m
  - ✓ Third location probed: h(k, 2) = [h'(k) + 2] mod m, and so on

LINEAR PROBING

Example: Consider a hash table of size 10. Using linear probing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table.

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Solution: Hash Function h(k, i) = [k mod m + i] mod m, Here m = 10.

Key	Location
72	2
27	7
36	6
24	4
63	3
81	1

LINEAR PROBING

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Key	Location
92	2
101	1

- For key 92, as location 2 already contains data value, we probe next locations sequentially untill we get empty slot or reach at the end of table.
- h(92, 1) = [92 mod 10 + 1] mod 10 = [2 + 1] mod 10 = 3 ..... Not empty
- h(92, 2) = [92 mod 10 + 2] mod 10 = [2 + 2] mod 10 = 4 ..... Not empty
- h(92, 3) = [92 mod 10 + 3] mod 10 = [2 + 3] mod 10 = 5 ..... Empty
- We can insert 92 at 5th position.
- What about 101???

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	92	36	27	-1	-1

QUADRATIC PROBING

- A variation of the linear probing.
- Instead of using a constant "skip" value, we use a rehash function that increments the hash value by 1, 4, 9 and so on. This means that if the first hash value is h, the successive values are h+1, h+4, h+9, h+16, and so on.
- In general, the i will be i^2, rehash(pos) = (h + i^2). In other words, quadratic probing uses a skip consisting of successive perfect squares.
- h(k, i) = [h'(k) + i^2] mod m,

  Here h'(k) = h(k) mod m

  m is the size of the hash table

  i is the probe number that varies from 0 to m–1
- Probe sequence:
  - ✓ First location probed: h(k, 0) = [h'(k) + 0] mod m
  - ✓ Second location probed: h(k, 1) = [h'(k) + 1] mod m
  - ✓ Third location probed: h(k, 2) = [h'(k) + 4] mod m, and so on

QUADRATIC PROBING

Example: Consider a hash table of size 10. Using linear probing, insert the keys 72, 27, 36, 24, 63, 81 and 101 into the table.

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Solution: Hash Function $h(k, i) = [k \bmod m + i^2] \bmod m$, Here $m = 10$.

Key	Location
72	2
27	7
36	6
24	4
63	3
81	1

QUADRATIC PROBING

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Key	Location
101	1

- For key 101, as location 1 already contains data value, we probe quadratic locations untill we get empty slot or reach at the end of table.
- h(101, 1) = [101 mod 10 + 1^2] mod 10 = [1 + 1] mod 10 = 2 ..... Not empty
- h(101, 2) = [101 mod 10 + 2^2] mod 10 = [1 + 4] mod 10 = 5 ..... Empty
- We can insert 101 at 5th position.

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	101	36	27	-1	-1

THANK YOU