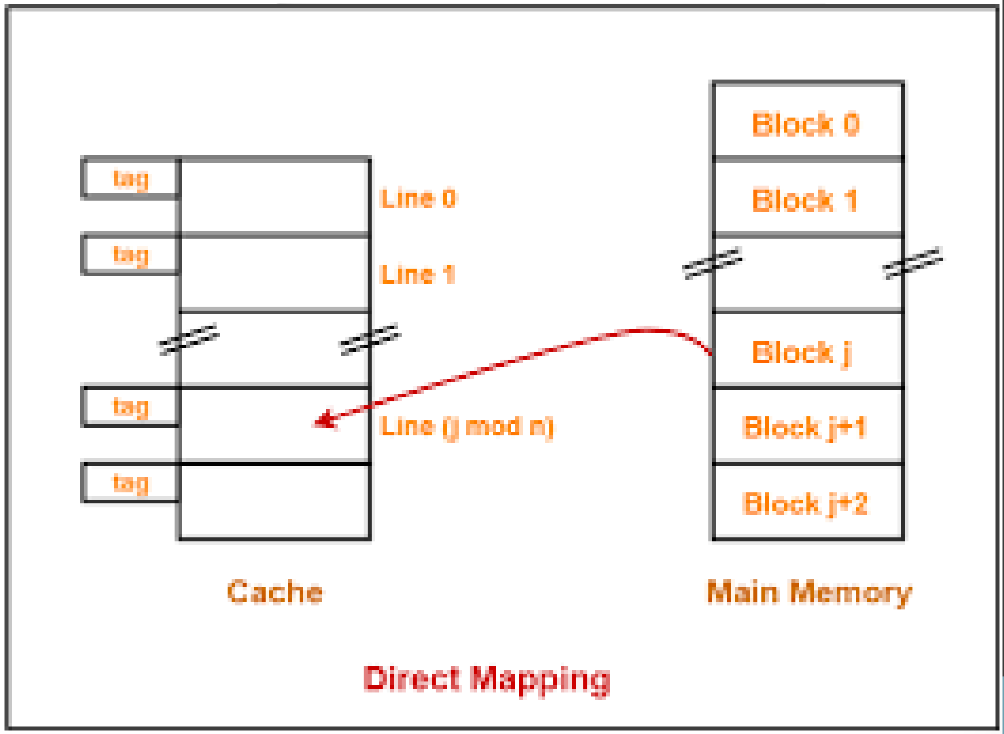
Data Structure and Algorithm

Hashing

Kanak Kalyani

Concept of Hashing

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



Hash Table

- 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).

Ideal Hashing

- Uses an array table[0:b-1].
  - Each position of this array is a bucket.
  - A bucket can normally hold only one dictionary pair.
- Uses a hash function f that converts each key k into an index in the range [0, b-1].
- Every pair (key, element) is stored in its home bucket table[f[key]].

3		**22**	**33**		**57**	**69**
[0]	[1]	[2]	[3]	[4]	[5]	[6]

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.

Hash Functions

- Division
- Middle of square
- Multiplicative
- Folding

Hashing By Division

- Domain is all integers.
- A key is mapped into one of m slots using the function

  $h(k) = k \bmod m$
- The division method results in a uniform hash function that maps approximately the same number of keys into each bucket.

Calculate the hash values of keys 1234 and 5462.

Setting M = 97, hash values can be calculated as:

- h(1234) = 1234 % 97 = 70
- h(5642) = 5642 % 97 = 16

Hashing by Multiplication Method

Step 1: Choose a constant A such that 0 < A < 1.
Step 2: Multiply the key k by A.
Step 3: Extract the fractional part of kA.
Step 4: Multiply the result of Step 3 by the size of hash table (m).

Example:
Given a hash table of size 1000, map the key 12345 to an appropriate location in the hash table.
Solution : *Let us consider* A = 0.618033, m = 1000, and k = 12345
h(12345) = ⌊ 1000 (12345 * 0.618033 mod 1) ⌋
h(12345) = ⌊ 1000 (7629.617385 mod 1) ⌋
h(12345) = ⌊ 1000 (0.617385) ⌋
h(12345) = ⌊ 617.385 ⌋
h(12345) = ⌊ 617⌋

Hashing by Mid Square Method

Step 1: Square the value of the key. That is, find k2.

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

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

Solution: hash table has 100 memory locations → 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 = 15**22**756, h (1234) = 27

When k = 5642, k^2 = 318**32**164, h (5642) = 21

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

Kanak Kalyani

Hashing by folding method

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.

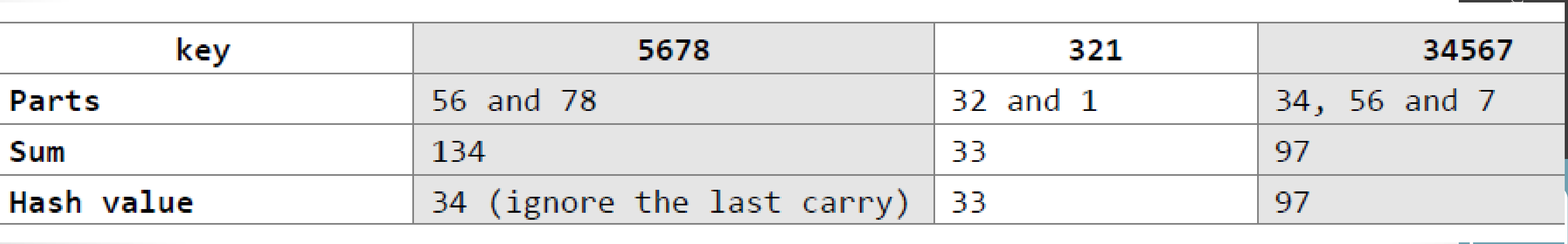
Step 2: Add the individual parts. That is, obtain the sum of k1 + k2 + ... + kn. The hash value is produced by ignoring the last carry, if any.

Hashing by folding method

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

Collisions

When 2 different keys are mapped to same location it is known as collision

Collision Resolution Techniques

1. <u>**Open addressing**</u>
   1. Linear Probing
   2. Quadratic Probing
   3. Double Hashing
2. Chaining

Open Addressing: Linear Probing

- Hash Table Size = 17.
- Slot = key % 17.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
34	0	45				6	23	7			28	12	29	11	30	33

Insert the keys: 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

Open Addressing: Linear Probing

Hash Function : h(k, i) = [h'(k) + i] mod m

6, 12, 34, 29, 28, 11,

Insert key 6: h(6,0) = [6 + 0]%17 = 6

Insert key 12: h(12,0) = [12 + 0]%17 = 12

Insert key 34: h(34,0) = [34 + 0]%17 = 0

Insert key 29: h(29,0) = [29 + 0]%17 = 12

h(29,1) = [29 + 1]%17 = 13

Insert key 28: h(12,0) = [28 + 0]%17 = 11

Insert key 11: h(11,0) = [11 + 0]%17 = 11

h(11,1) = [11 + 1]%17 = 12

h(11,0) = [11 + 2]%17 = 13

h(11,0) = [11 + 3]%17 = 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
34	0	45				6	23	7			28	12	29	11	30	33

Disadvantage: Primary Clustering

Insert 17,34,51, 68, 85,102, 119

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	34	51	68	85	102	119										

Open Addressing: Quadratic Probing

Hash Function: $h(k, i) = [h'(k) + c_1 * i + c_2 * i^2] \bmod m$

- Hash Table Size = 17.
- Slot = key % 17.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
34	0	45				6	23	7			28	12	29	30	11	33

Insert the keys: 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

28-10-2024

Open Addressing: Quadratic Probing

Hash Function: $h(k, i) = [h'(k) + c_1*i + c_2*i^2] \bmod m$

Consider $c_1=0$ $c_2=1$

6, 12, 34, 29, 28, 11,

Insert key 6: $h(6,0) = [6 + 0]\%17 = 6$

Insert key 12: $h(12,0) = [12 + 0]\%17 = 12$

Insert key 34: $h(34,0) = [34 + 0]\%17 = 0$

Insert key 29: $h(29,0) = [29 + 0]\%17 = 12$

$h(29,1) = [29 + 1^2]\%17 = 13$

Insert key 28: $h(12,0) = [28 + 0]\%17 = 11$

Insert key 11: $h(11,0) = [11 + 0]\%17 = 11$

$h(11,1) = [11 + 1^2]\%17 = 12$

$h(11,0) = [11 + 2^2]\%17 = 15$

Disadvantage: Secondary Clustering

Secondary Clustering is the tendency for a collision resolution scheme such as quadratic probing to create long runs of filled slots *away* from the hash position of keys.

- If the primary hash index is x, probes go to x+1, x+4, x+9, x+16, x+25 and so on, this results in Secondary Clustering.
- Secondary clustering is less severe in terms of performance hit than primary clustering, and is an attempt to keep clusters from forming by using Quadratic Probing.

Rehashing/ Double Hashing

- **Rehashing**: Try H_1, H_2, …, H_m in sequence if collision occurs. Here H_i is a hash function.
- **Double hashing** is one of the best methods for dealing with collisions.
  - If the slot is full, then a second hash function is calculated and combined with the first hash function.
  - $H(k, i) = (H_1(k) + i\, H_2(k)) \% m$

The Second Hash Function should be

- Quick to evaluate
- Never evaluate to zero
- Different from previous hash function

Example: Double Hashing

$h_1(k) = k \bmod 13$

$h_2(k) = 1 + (k \bmod 11)$

$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod 13$

- Insert key 14:

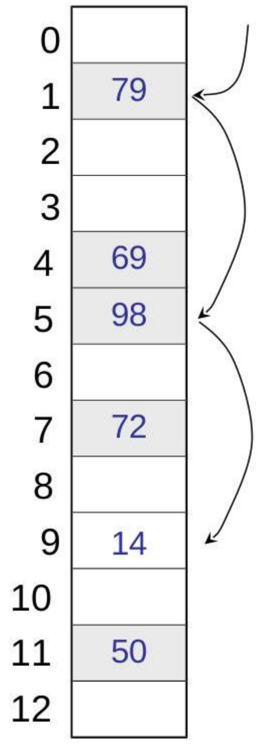
$h_1(14,0) = 14 \bmod 13 = 1$

$h(14,1) = (h_1(14) + h_2(14)) \bmod 13$

$= (1 + 4) \bmod 13 = 5$

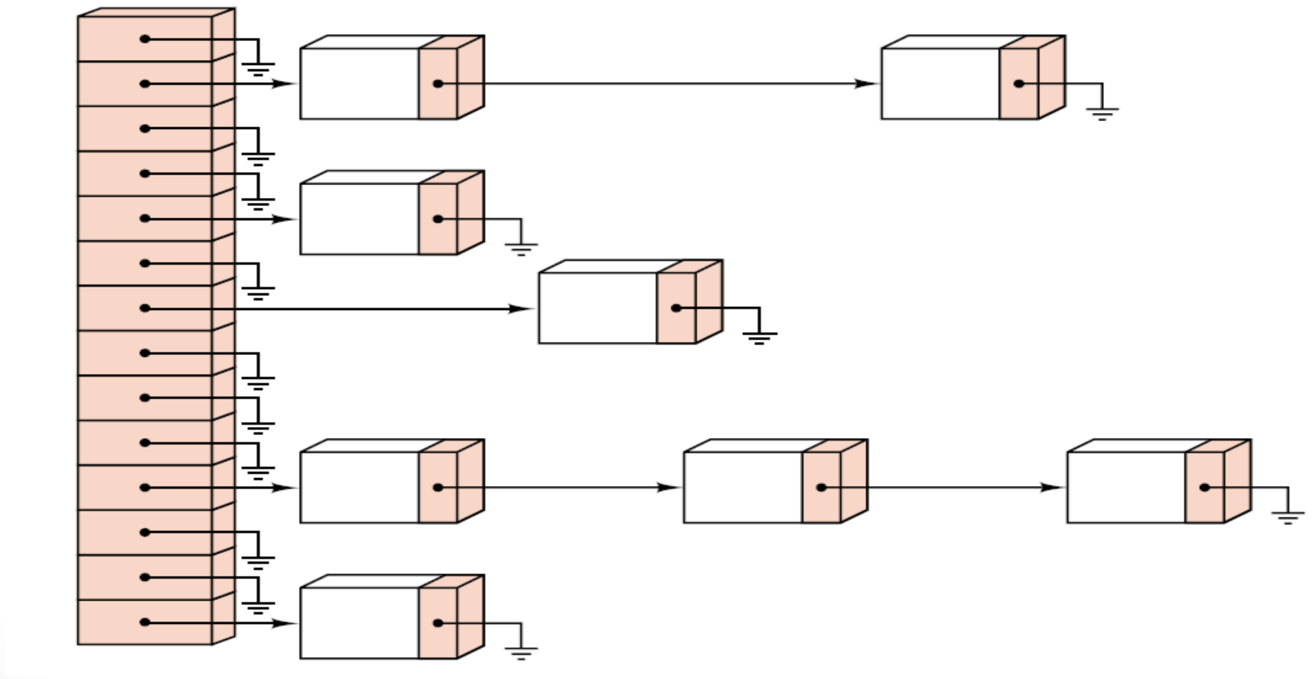
$h(14,2) = (h_1(14) + 2\, h_2(14)) \bmod 13$

$= (1 + 8) \bmod 13 = 9$



Collision Resolution by Chaining

- In chaining, each location in a hash table stores a pointer to a linked list that contains all the key values that were hashed to that location



Chaining Example

Example: Load the keys **23, 13, 21, 14, 7, 8, and 15** , in this order, in a hash table of size **7** using separate chaining with the hash function: **h(key) = key % 7**

h(23) = 23 % 7 = 2

h(13) = 13 % 7 = 6

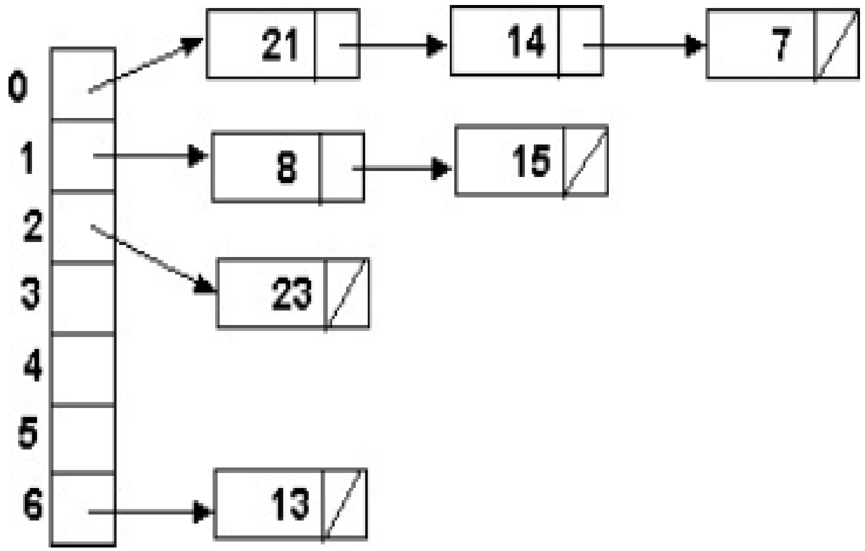
h(21) = 21 % 7 = 0

h(14) = 14 % 7 = 0 collision

h(7) = 7 % 7 = 0 collision

h(8) = 8 % 7 = 1

h(15) = 15 % 7 = 1 collision



Kanak Kalyani

Hash Table Performance

- If No Collisions, Searching takes O(1)
- MORE Collisions ➔ Performance Decrease
- Why do we have more collisions??
    - If number of keys are more ➔ more collisions can be there as they may map to same location
- The solution to the problem is to choose an appropriate size of table.
    - **The *size* of the hash table is the actual number of elements in the table; the *capacity* of the table is the number of components that it has. The ratio of these two parameters is called the *load factor*.**

Load Factor

- Consider number of elements in hash table was 6 and the capacity was 101, resulting in a load factor of 6/101 = 5.94%. If elements in hash table are 100, resulting in a load factor of 100/101 = 99.01%.

- Let N = number of items to be stored
- Load factor(λ) = N/TableSize

- Optimal Load Factor should be below 0.75