CHAPTER 6- HASHING

LEC 1: INTRODUCTION

DATE-

* Main usage of Datastructure is to store our data.

* Operations performed on datastructures:
  - Insertion
  - Searching (most dominant operation) — Needs to be faster
  - Deletion

		Time Complexity (to search)
* Array	–	O(n)
* Sorted Array	–	O(log n)
* Linked List	–	O(n)
* Binary Tree	–	O(n)
* Binary Search Tree	–	~~O(log n)~~ O(n)
* Balanced Binary Search Tree	–	O(log n)
* Priority queues (min heap & max heap)		O(n)

Note:
Hashing → solution to reduce search Time
↳ Time Complexity - O(1) i.e constant time

Direct Address Table → (11) to arrays, was used before Hashing
↳ 1st data structure with O(1) time complexity for searching

100 records have to be stored, each should have unique tag {in rage from 1 to 100}

define array of 100 size, store / insert elements at appropriate slot
{1, 2, 3, ----, 100}

↳ Can be used only if Number of keys used are proportional to size of array.

* If number of keys to be inserted are very less but size of each key is very large, then this method is not useful eg:- {1000 0001, 1000 0002, 1000 0005}
even to insert these array index has to be this large

LEC 2: HASHING INTRO

DATE-

Problem with DAT:

The size of the array / table should be as big as the key sizes that are to be stored

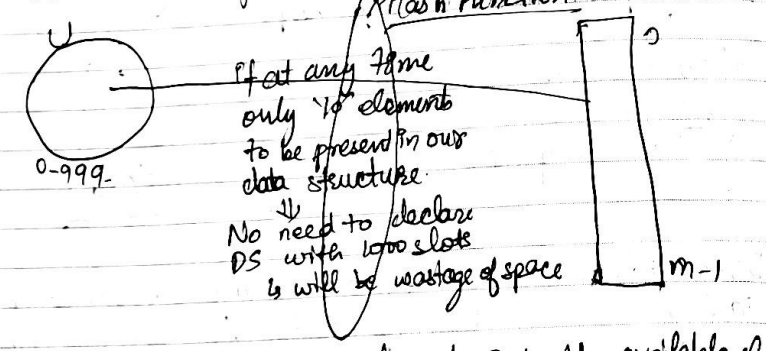
↳ Sometimes we end up declaring size of array as too large even if number of elements to be stored are very less.

HASHING

{ means taking a set of very large numbers and mapping it to small numbers which correspond to indices of hash table.

* Even though keys to be stored are very large but at any time the numbers of elements to be stored in that are less

↳ we declare a data structure - large enough to store elements which might be present at any point in time.

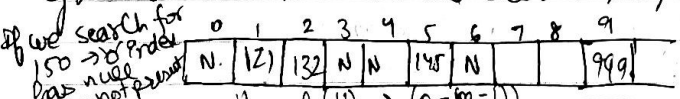


Then we try to map elements into the available slots.

Hash Function: Mapping function that is used to map the 'key' (to be stored from Universal set) to the location at which it will be stored in available data structure.

Also applied on element at the time of searching & check its availability at the said location in data structure.

Eg:- Elements to be Inserted are: 121, 145, 132, 999 (0-999)



Hash Function (0-999) → (0-9) ⇒ h(val) = mod 10

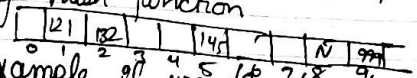
↳ which will take values from (0-999), map them to values (0-9)

h(val) = mod 10

DATE-

PROBLEM WITH HASHING: COLLISION

When more than one elements map to same location in hash table on applying hash function



In prev example if we try to store 131 then it will lead to collision

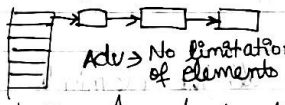
SOLUTION FOR COLLISION i.e COLLISION RESOLUTION TECHNIQUE:

(I) Make Hashing function better.

* We are considering only least significant bit in previous example.

↳ Problem:
We can have better hashing function which will decrease the number of collisions but they can't completely vanish.

(II) CHAINING



Adv → No limitation on number of elements to be saved.

↳ says in Hash table instead of storing elements directly at its indices, maintain a linked list corresponding to each index in which elements will be stored.

* More than one elements can be accomodated at same index in linked list.

↳ Space wasted on pointer. ↳ Used → when insertion & deletion both involved, searching

(III) OPEN ADDRESSING or Probing
searching for empty space in which we want to store element.

↳ If collision occurs, recompute the hash value with some minor modification to hash value - might lead to vacant space (perform recomputation till we get some vacant space)

a) Linear Probing
b) Quad "
c) Double "

only Insertion & Search
→ Used - when no deletion involved
→ Disadvantage - No. of elements to be saved are limited by size of array.

LEC 3: CHAINING

* If we want to insert an element or key 'K' then we have to go to location T[h(K)]
  ↳ Hash function - give index
  ↓
* Instead of placing element at the index given by hash function.
* Rather element is inserted into the list pointed by the table's location (T[h(K)])
  ↓
  New node is created and inserted at the beginning of the list.

Insertion Time complexity = O(1) {∵ computing Hash func^n & inserting at beggining of linked list takes constant time}

Worst case scenario of ~~Insertion~~ storing elements: (searching)
↳ When all the elements hash function returns same index
↳ All the element gets stored in linked list present at one index only.

Searching Time complexity = O(n) {∵ of this worst case}

Worst case for deletion:
↳ Search for the element and then delete it.

Deletion Time complexity = O(n)

Even these worst case complexities of O(n) can be avoided by using Uniform Hash Function.

Uniform Hash function (Uniform distribution of hashing function)
↳ Which uniformly distributes the elements in the array
↳ Doesnot map all elements to same location

If Uniform Hash Function used: (Avg Case for Searching)

each location of array will get → $\left(\frac{n}{m}\right)$ elements in every DATE- at various location

Average Search Time Complexity → $\Theta = \left(1 + \left(\frac{n}{m}\right)\right)$ ← to compute hash function

m - size of hash table

Load factor $\alpha = \frac{n}{m}$ (n: Total no. of element, m: Total size of hash table)

If $n = Km \Rightarrow \Theta = (1 + Km) = \Theta(1)$ → Avg Case.
↳ Avg Deletion → Θ(1)

Advantage of chaining :→ In Avg case → most of the operations take const time.
↳ Deletion is easy → In order to delete an element, no need to affect the remaining elements.

Disadvantage :-
↳ Pointers (space is wasted)

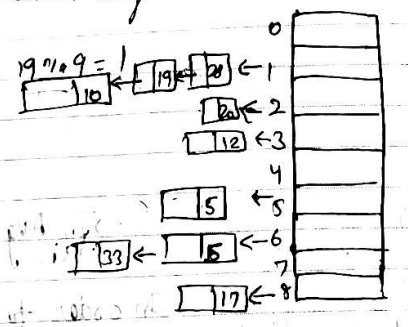
LEC 4: EXAMPLE

DATE-

Gate 96) An advantage of chained hash table over open addressing scheme is
a) Worst case complexity of search is less
b) Space used is less
c) Deletion is easier.
d) None of the above.

Deletion is easier if we use chained hash table.

Gate 2014
$h(K) = K \bmod 9$, hash table has 9 slots, chaining is used.
Keys: 5, 28, 19, 15, 20, 33, 12, 17 and 10.
Then max, min & avg chain lengths in hash table.



$5 \%9 = 5$, $28 \%9 = 1$, $19 \%9 = 1$
$15 \%9 = 6$, $33 \%9 =$
$20 \%9 = 2$, $12 \%9 = 3$
$17 \%9 = 8$

min chain length = 0
max chain " = 3
avg " " $= \alpha = \frac{n}{m} = \frac{9}{9} = 1$

Gate 2014
Consider a hash table with 100 slots. Collisions are resolved using chaining. Assuming simple uniform hashing, what is the probability that the first 3 slots are unfilled after 1st 3 insertions.

→ Uniform distribution of elements using hashing.

probability that particular key 'K' is mapped to one of the locations in array $= \frac{1}{n}$ [For every slot]

Required probability $= \left(97 \times \frac{1}{100}\right)\left(97 \times \frac{1}{100}\right)\left(97 \times \frac{1}{100}\right)$

Gate 97 Consider a hash table with 'n' buckets, where chaining is used to resolve collisions. The hash function is such that the probability that a key value is hashed to a particular bucket is $\frac{1}{n}$. The hash table is initially empty and 'k' distinct values are inserted in the table.

a) What is the probability that bucket number '1' is empty after 'K' insertion.
b) What is the probability that no collision has occurred in any of the 'K' insertions?
c) What is the probability that 1st collision occur at the K^{th} insertion?

Size of hash table = n
Probability that a key value is hashed to particular bucket $= \frac{1}{n}$

a) Prob that bucket number 1 is empty after 'K' insertion $= \left(1 - \frac{1}{n}\right)^K \times K = \left(\frac{n-1}{n}\right)^K \times K$

b) Prob that no collision has occurred in any of the 'K' insertions $= \left(\frac{n}{n}\right)\left(\frac{(n-1)}{n}\right)\left(\frac{n-2}{n}\right) \cdots \left(\frac{n-(k-1)}{n}\right)$

c) Prob that 1st collision occur at K^{th} insertion
$= \left(\frac{n}{n}\right)\left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right) \cdots \underbrace{\left(\frac{n-(K-2)}{n}\right)}_{(K-1)^{th} \text{ element}} \left(\frac{K-1}{n}\right)$

K^{th} element has to take a slot taken by (K-1) elements.

LEC-7: OPEN ADDRESSING

DATE-

OR CLOSED HASHING

* All the elements are stored in slots available in the array not in linked list outside array.

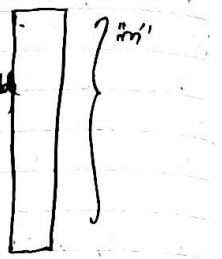
↳ Consequences - The number of elements stored are restricted by size of array.

Load factor (α) = $\frac{\text{Total no. of elements present in the hash table}}{\text{Total size of the hash table}}$

max load factor = $\frac{m}{m} = 1$

min " " = 0

$0 \le \alpha \le 1$



* Number of elements or keys to be stored can be much more than the number of slots available in array.

Collision possible

↳ In this case also elements are to be stored in same hash table

↳ Collision resolved by again applying a hash function with minor modification

↓

called 2nd time Probing (finding a location in hash table at which element is to be stored)

$h: U \to \{0, 1, 2, \dots, m-1\}$ If $h(k)$ already filled.

2nd time probing $h: (U \times \{0, 1, 2, \dots, m-1\}) \to \{0, 1, 2, \dots, m-1\}$

↳ taking already filled locations into consideration.

Using same key & in second set increase the numbers by 1 (again run hash func") if 1st collision then if 2nd time collision occurs " " 2 & again run hash func"

↓

Hash function is not only a function of key but also a function of key + collision number or probing number

DATE-

* In case of LHS in the function → every location of hash table is inspected atleast once.

↳ Probing Sequence - The sequence of examination/inspection of locations of hashtable done in order to insert an element into the hash table is known as probing sequence.

Worst time Complexity for searching :- $O(m)$

↳ In this case, if element not present at some slot then we assume that it can be at some other slot.

↓

We can be lucky if either we find the element on the way or we find a NULL. → searching ends there. Can't stop on seeing d (space created after deletion)

$h(K) = 1$
$h(K_1) = 2$ (already occupied)
$h(K_1, 1) = 6$ (" ")
$h(K_1, 2) = 3$ (Null)

0	N
1	K_1
2	X
3	K_2
4	X d
5	N
6	X
7	N

Searching also done in similar sequence.

For searching we will be computing hash func" for K_1 in following sequence.

$f(K_1, 0) = 2$
$f(K_1, 1) = 6$
$f(K_1, 2) = 3$
$f(K_1, 3) = 4$
$f(K_1, 4) = 5$
$f(K_1, 5) = 1$
$f(K_1, 6) = 7$
$f(K_1, 7) = 0$

Before declaring that element can't be inserted or not present in hash table (in case of searching) each index of array is examined atleast once

Note: Deletion affects searching, but doesn't affect insertion.

whereas in case of Searching, 'd' considered as OCCUPIED

* In case of Insertion, 'd' considered as NULL

* In case of Deletion, we will place some other marked d (instead of NULL) to indicate that space is available after insertion. like d

'd' considered to be occupied.

DATE-

Worst case scenario after Deletion:

↳ If all the cells are free after deletion or are NULL & only 1 element present in hash table, search will take a lot of time

0	N
1	N
2	D
3	D
4	D
5	K1
6	D
7	N

If 'K1' is present at last probe, then inspite of empty hash table we have to check entire table.

Note:- If Deletion is involved, Chaining is much better than Open Addressing.

Adv over Chaining:-

↳ Space not wasted for pointer.

LEC8: METHODS OF OPEN ADRESSING

LINEAR PROBING: (probing location in linear manner)

* We have to just modify hash function to make it work as linear probing.

* We have following hash function:-

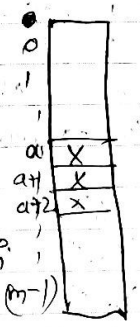
$h: U \rightarrow \{0, 1, 2, \dots, m-1\}$

If $h(K) = a$ → filled → prob. again

hash function is going to take a key and apply hash function to get location at which we in hash table.

$$h'(K, i) = (h(K) + i) \bmod m$$

→ Hash function for linear probing



Why linear?

From 1st location, all the locations - linearly increasing are probed till the end then from 1st location till the 1st probed location.

DATE-

$h(K) = a$

$h'(K, 1) = (h(K) + 1) \bmod m$

$= (a+1)$ → filled

$h'(K, 2) = (h(K) + 2) \bmod m$

$= a+2$

Probing sequence:-

The sequence of inspecting/examining locations of hash table before finally storing that element.

eg:- If sequence is: $(0, 1, 2, \dots, m-1)$] If 'm' slots in hash table, there are every slot has to be probed exactly once.

↓

Number of probs required = m

$h'(K, i)$ → ensure probing of all locations

can be:

$(1, 2, \dots, m-1, 0)$

$(2, 3, \dots, m-1, 0, 1)$

Number of probing sequences = m (LINEAR PROBING)

Note:- If two numbers have initial probe number or location to be same then they will have same probe sequence. } called secondary clustering.

↳ All 'm' probs will generate different slots.

* Hash function chosen for linear probing should examine each slot of the hash table.

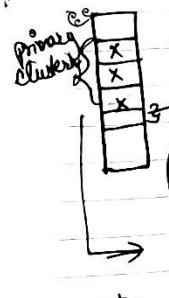
* Problem with linear Probing: (1) Secondary Clustering (2) Primary Clustering

* In case of linear probing, there will be continous runs of filled slots

* Means many filled slots items will fall together in continous run, then a gap, again continous filled slots.

↳ increasing search time.∴ we have to search through entire cluster

1/m → probability for each slot in case of uniform distribution



→ probability that this empty slot will get an element?

Either element directly gets mapped to this location
or If that particular key maps on to one of the slots of primary cluster.

4/m

Note #

If 'i' slots are in primary clustering then probability that (i+1)th slot will get filled is :

$\left(\frac{i+1}{m}\right)$

Avg. Search Time → O(1) → Practically or experimentally

* Summary of linear probing
  - Will have 2 hash function → Normal Hashing func^n
    → Probing " "
  - If initial slot is full, the sequence which we follow to insert an element is known as Probing Sequence.
  - Probe sequence depends on initial probe number.
  - 2 disadvantages → Primary Clustering
    → secondary "

Why called linear probing?

$h'(K,i) = (h(K) + i) \bmod m$

Because if collision occurs at location h(K) then the next location to be probed is a linear distance from initially probed location.

LEC 9: EXAMPLES ON LINEAR PROBING

Gate 2008

Keys: 12, 18, 13, 2, 3, 23, 5, 15

Hash Table Size = 10

h(K) = K mod 10 & linear probing

What is the resultant hash table?

$h'(K,i) = (h(K) + i) \bmod 10$

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

h(12) = 12 %. 10 = 2

h(18) = 18 %. 10 = 8

h(13) = 13 %. 10 = 3

h(2) = 2 %. 10 = 2 (already occupied)
h(2,1) = (2+1) %. 10 = 3 (" ")
h(2,2) = (2+2) %. 10 = 4

h(3) = 3 %. 10 = 3
h(3,1) = 4 %. 10 = 4
h(3,2) = 5 %. 10 = 5

h(23) = 23 %. 10 = 3
h(23,1) = 24 %. 10 = 4
h(23,2) = 25 %. 10 = 5
h(23,3) = 26 %. 10 = 6

h(5) = 5 %. 10 = 5
h(5,1) = 6 %. 10 = 6
h(5,2) = 7 %. 10 = 7
h(15) = 15 %. 10 = 5

h(15,1) = 16 %. 10 = 6
h(15,2) = 17 %. 10 = 7
h(15,3) = 18 %. 10 = 8
h(15,4) = 19 %. 10 = 9

DATE-

Gate IT-08

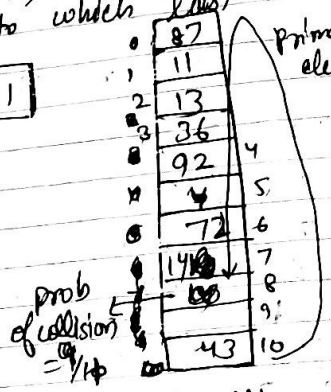
Hash Table size = (0-10)

$h(K) = K \bmod 11$, linear probing used.

Keys: 43, 36, 92, 87, 11, 4, 71, 13, 14

What is the index into which last record is inserted?

$h(K, i) = (h(K) + i) \bmod 11$



$h(43) = 43 \% 11 = 10$

$h(36) = 36 \% 11 = 3$

$h(92) = 92 \% 11 = 4$

$h(87) = 87 \% 11 = 10$

$h(87, 1) = 88 \% 11 = 0$

$h(11) = 11 \% 11 = 0$

$h(11, 1) = 12 \% 11 = 1$

$h(4) = 4 \% 11 = 4$

$h(4, 1) = 5 \% 11 = 5$

$h(71) = 71 \% 11 = 5$

$h(71, 1) = 72 \% 11 = 6$

$h(13) = 13 \% 11 = 2$

$h(13, 1) = 14 \% 11 = 3$

$15 \% 11 = 4$

$= 5$

$= 6$

~~$h(13, 5) = 18 \% 11 = 7$~~

$h(14) = 14 \% 11 = 3$

4

5

6

$h(14, 6) = 18 \% 11 = 7$

prob of getting next element $= \frac{10}{11}$

After getting next element $= \frac{1}{11}$

Last record

Last index filled = 8

Create 2016 $h(K) = K \bmod 10$

Linear Probing

DATE-

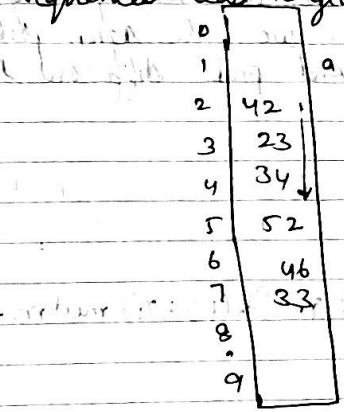
5) Which of the following sequences lead to give hash table

a) 46, 42, 34, 52, 23, 33

b) 34, 42, 23, 52, 33, 46

c) 46, 34, 42, 23, 52, 33

d) 42, 46, 33, 23, 34, 52



(P) How many insertion sequences are possible?

a) 46-6, 42-2, 34-4, 52-8 ⇒ ✗

b) 34-4, 42-2, 23-3, 52-2-3-4-5 ✓, 33-6 ✗

✓c) 46-6, 34-4, 42-2, 23-3, 52-2-3-4-5, 33-3-4-5-6-7

d) 42-2, 46-6, 33-3 ✗

(42, 23, 34) → 3! ways, 52 has to come after these 3

33 → after 52

(42, 23, 34) 52 33

fixed

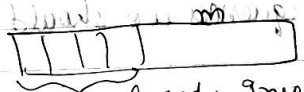
5 places for 46

∴ No. of insertion sequence = 3! × 5 = 30 way

Gate IT-07

Consider a hash function that distributes keys uniformly. The Hash table size is 20. After hashing of how many keys will the probability that any new key hashed collides with an existing one exceed 0.5?

a) 5 b) 6 c) 7 d) 10



i already inserted

prob of collision for next element $= \frac{i}{m}$

prob of collision $= \frac{5}{20} = \frac{1}{4} = 0.25$

$\frac{6}{20} = \frac{3}{10} = 0.3$

$\frac{7}{20} < 0.5$

$\frac{10}{20} = 0.5$

Ans = 11 ⇒ (d)

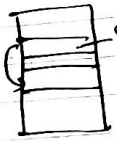
LEC 13: QUADRATIC PROBING

DATE-

* Similar to linear probing which means in case of collision we will again probe.
* For second probe different hash function will be used.

Hash function for Linear Probing:
$h'(K,i) = (h(K)+i) \bmod m$ → can be modified as

$h'(K,i) = (h(K)+2i) \bmod m$ ⇒



Hash Function for Quadratic Probing:
$h'(K) = (h(K) + C_1 i + C_2 i^2) \bmod m$ ⇒

Boobo

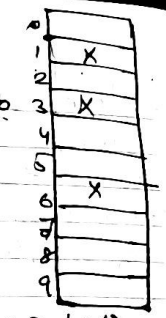
* Distance of probe position from initial probe position increases quadratically.
* Advantage of Quadratic probing over linear probing:
  ↳ Doesnot suffer with Primary Clustering ∵ probability that elements will be stored contiguously is reduced in this case.

Secondary clustering possible
↳ i.e if initial probe position is same for any 2 keys then the entire probing sequence will also be same for both.

* By doing only 'm' probe sequences we should be able to examine the entire table.
* If we choose C_1, C_2, m randomly then problem may arise.

DATE-

eg: $m = 10$ (0-9), $C_1 = 1$, $C_2 = 1$, $h(K) = K \% 10$



Let $h(K_1) = 1$ (already occupied)

$h'(K_1, 1) = h(K_1) + ? + i^2$ ← same

$= 1 + 1 + 1 = 3$ (also occupied)

$h'(K_1, 2) = 1 + 1 + 2^2 = 6$

$h'(K_1, 3) = 1 + 1 + 9 = 11 \% 10 = 1$ (already probed)

Should Not happen in general

$h'(K_1, 4) = 1 + 1 + 16 = 8$

$h'(K_1, 5) = 1 + 1 + 25 = 7$ } same

$h'(K_1, 6) = 1 + 1 + 36 = 7$ }

* Each probe in probe sequence should give unique / different probe location.

↳ Problem in Quadratic probing
we might not be able to examine 'm' nodes / slots of hash table using 'm' probing -
we might end up assuming that the table is full even without examining some of the location.

Note :- C_1, C_2, m should be chosen in such a way that whatever the initial probe position is, if 'm' probes are done - one should be able to probe / examine the entire table

↳ Coming up with such numbers is challenging

↳ Note - Due to above mentioned constraints of choosing C_1, C_2, m wisely - Quadratic probing is not preferred
More commonly used method is Linear probing ∵ of simplicity.

Note :- Secondary clustering is possible ∵ only initial probe is dependent on key, later probes are independent of key
↳ problem with both linear and quadratic probing

DATE-

Q How many different probe sequences are possible using quadratic probing? ⇒ m :: m - initial probe ~~seq~~ are possible

Note: Number of probe sequences depend on number of initial probe sequence.

Note: Number of collisions can't be reduced unless we have lot of probe sequences.

LEC14: DOUBLE HASHING

* Instead of having one hash function & then constant displacement, Here we will have 2 hash functions.

Hash Function

$h'(K) = (h_1(K) + i\,h_2(K)) \bmod n$

↓

Advantage of new hash func^n

↳ No primary clustering (:: probe displacement is no way linear {:: next probe position decided by $h_2(K)$})

Note:- In Double hashing, successive probes donot depend on initial probe as probe sequence can vary even in case of same initial probe.

↳ No secondary clustering.

Number of probe sequences possible $= \theta(m^2)$

↳ Reason: Every probe can start at some position from $(0-(m-1))$ can have displacement varying from $(0-(m-1))$

DATE-

Number of hash functions to be used to decide displacement can be decreased, decreasing the chances of collision

↑ Not widely used

Note:- Double hashing is best known technique, $h_2(K)$, m should be decided carefully so that all the hash table slots are examined

↳ try to make them relatively prime ($h_2(K)$, m)

odd & $< m$ or 2 & prime

Note:- Symbol table in compilers is a hash table

Note:- Hashing preferred if major operation is searching, data is static