Assignment No. 6.

Title :- Linear probing with & without replacement.

Aim :- To learn & implement the hashing with linear probing with & without replacement.

Problem Statement :- Implement all the functions of dictionary using hashing. Data: set of (key, value) pair. Keys are mapped to values. Key must be comparable, keys must be unique. Standard operation: insert(key), Find(key), Delete(key) (use linear probing with & without replacement calculate average search cost for both.)

Objective :- i) To learn & implement the hashing data structure.
ii) To understand the use of hashing & advantage of hashing over tree
iii) To understand different hashing functions with different collision resolution strategies.

Theory :-

Hashing :- Hashing is an important data structure is designed to use a special function called hash function which is used to map a given value with a particular key for faster access of elements. The efficiency of hashing mapping depends on the efficiency of hash function used.

Let hash function $H(x)$ maps the value at index $x \% 10$ in an array. for e.g. the values {11, 12, 13, 14, 15} will be stored at position {1, 2, 3, 4, 5} in hash table.

• Collision in hashing -

- In hashing, hash function used to compute hash value to for a key.
- Hash value is then used as an index to store the key.
- Hash function may return same hash value for two or more keys. It means the collision occurs.
- A collision occurs when two non-identical identifiers are hashed to same bucket.

• Collision resolution strategy :-

i) Linear probing -

In linear probing, when collision occurs we linearly search for next bucket. We keep probing until an empty bucket is found.

- With replacement -

In this we linearly search for next bucket if collision occurs.

e.g. 10, 15, 14, 20, 40

hash function = h(x) = $x \% 5$

0	10
1	15
2	20
3	40
4	14

$10 \% 5 = 0$
$15 \% 5 = 0$
$20 \% 5 = 0$
$40 \% 5 = 0$
$14 \% 5 = 4$

· Linear probing with replacement :-

In this type of strategy, if collision occur then we linearly search for next available bucket & store the value in it. If hash value is calculated & the incorrect value is stored in the bucket then correct value is stored in appropriate bucket & previous value is stored in next available bucket.

e.g. 13, 23, 20, 14, 10

0	20
1	23
2	10
3	13
4	14

$20 \% 5 = 0$
$23 \% 5 = 3$
$10 \% 5 = 0$
$13 \% 5 = 3$
$14 \% 5 = 4$

Operations :

i) Inseet (key, value) :

First hash value of key is calculated & then data is inseeted at that hash address in hash table.

0	10
1	
2	12
3	23

Before inseeting 31

$h(x) = x \% 10$

$h(x) = 31 \% 10$

1

0	10
1	31
2	12
3	23

After inseeting 31

ii) Deletion :-

Deletion of key from hash table required to calculate hash address & value to be deleted & making that bucket empty.

0	10
1	31
2	22
3	43

Before deleting 22

$h(x) = x \% 10$

$h(22) = 22 \% 10$

$= 2$

0	10
1	31
2	
3	43

After deleting 22

iii) search (key)

Searching is very simple process in hashing. It is very fast as compared to other DS.

0	10
1	31
2	22
3	43

$h(x) = x \% 10$

$h(43) = 43 \% 10$

3

Calculate the hash Value

43 Found at 3

Searching 43

- Pseudocode / Algorithm

i) Insertion :

Step 1:) Read the key & value to be inserted., key & value.

Step 2:) Calculate hash address using hash function say hasaddr

Step 3:) IF (hashTable [hasaddr] is empty) then
store the key & value at hasaddr
else
3.1) Set one pointer to next index of has addr
$i = hashaddr + 1$
3.2) Set Flag to 0
while ((i<n) && (flag == 0))
{

CLASSMATE
Date: 66
Page:

```
if( hashTable [i] is empty) then
      insert key & value  at i & set flag to 1
else
      increment  i

}
3.3) set i=0
   while((i<hashAddr) && (flag==0))
      {
      if (hashTable [i] is empty)
         {
         insert key & value  at i
            & set flag to 1.
         }
      else
         increment  i
      }
4) END
```

2) Searching :

Step 1:) Read the key to be search
Step 2:) Calculate hash value of key, say hashAddr
Step 3:) if ([hashAddr] == key)
      return (hashAddr && key)
   else
      for i = hashAddr+1 to (i%n) != hashAddr

Scanned with CamScanner

```
        if (hashTable [i%n] -> key == key)
            return (i%n)
        end if
      i++
    end for
  end if
```

Step 4:) Return -1
Step 5:) Stop

3) Deleting :

Step 1:] Read the key to be deleted.
Step 2:] Search the key to be deleted & store its position, say k

Step 3.] If (k != empty)
make the hashTable[k] as empty.
else
print "key not found"

Step 4:] stop.

classmate
Date : 68
Page :

Conclusion:

i) hash tables are faster in most cases but general problem with hash tables there O(1) complexity is not guaranteed.

ii) For addition there is a point when table becomes full, then table needs to be enlarged. which requires moving all of its elements.

iii) Hash put the operation & get the operation of time complexity O(1) with assumption that key-value pair are well distributed across the buckets.

iv) Time complexity:

	Average case	worst case
search	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)