

NAME: ADWAIT S PURAO
UID: 2021300101
EXP NO. : 10
AIM: To insert elements in a Hash Table using the concept of linear probing to resolve the collisions

THEORY:

Hashing in data structures

What is Hashing?

Hashing is the process of mapping large amount of data item to smaller table with the help of hashing function.

Hashing is also known as Hashing Algorithm or Message Digest Function.

It is a technique to convert a range of key values into a range of indexes of an array.

It is used to facilitate the next level searching method when compared with the linear or binary search.

Hashing allows to update and retrieve any data entry in a constant time $O(1)$.

Constant time $O(1)$ means the operation does not depend on the size of the data.

Hashing is used with a database to enable items to be retrieved more quickly.

It is used in the encryption and decryption of digital signatures.

What is Hash Function?

A fixed process converts a key to a hash key is known as a Hash Function.

This function takes a key and maps it to a value of a certain length which is called a Hash value or Hash.

Hash value represents the original string of characters, but it is normally smaller than the original.

It transfers the digital signature and then both hash value and signature are sent to the receiver. Receiver uses the same hash function to generate the hash value and then compares it to that received with the message.

If the hash values are same, the message is transmitted without errors.

What is Hash Table?

Hash table or hash map is a data structure used to store key-value pairs.

It is a collection of items stored to make it easy to find them later.

It uses a hash function to compute an index into an array of buckets or slots from which the desired value can be found.

It is an array of list where each list is known as bucket.

It contains value based on the key.

Hash table is used to implement the map interface and extends Dictionary class.

Hash table is synchronized and contains only unique elements.

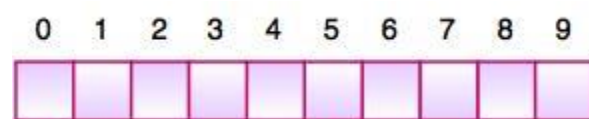


Fig. Hash Table

The above figure shows the hash table with the size of $n = 10$. Each position of the hash table is called as Slot. In the above hash table, there are n slots in the table, names = $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Slot 0, slot 1, slot 2 and so on. Hash table contains no items, so every slot is empty.

As we know the mapping between an item and the slot where item belongs in the hash table is called the hash function. The hash function takes any item in the collection and returns an integer in the range of slot names between 0 to $n-1$.

Suppose we have integer items $\{26, 70, 18, 31, 54, 93\}$. One common method of determining a hash key is the division method of hashing and the formula is :

Hash Key = Key Value % Number of Slots in the Table

Division method or reminder method takes an item and divides it by the table size and returns the remainder as its hash value.

Data Item	Value % No. of Slots	Hash Value
26	$26 \% 10 = 6$	6
70	$70 \% 10 = 0$	0
18	$18 \% 10 = 8$	8
31	$31 \% 10 = 1$	1
54	$54 \% 10 = 4$	4
93	$93 \% 10 = 3$	3

0	1	2	3	4	5	6	7	8	9
70	31		93	54		26		18	

Fig. Hash Table

After computing the hash values, we can insert each item into the hash table at the designated position as shown in the above figure. In the hash table, 6 of the 10 slots are occupied, it is referred to as the load factor and denoted by, $\lambda = \text{No. of items} / \text{table size}$. For example, $\lambda = 6/10$.

It is easy to search for an item using hash function where it computes the slot name for the item and then checks the hash table to see if it is present.

Constant amount of time $O(1)$ is required to compute the hash value and index of the hash table at that location.

Collision Resolution Techniques

Hashing in data structure falls into a collision if two keys are assigned the same index number in the hash table. The collision creates a problem because each index in a hash table is supposed to store only one value. Hashing in data structure uses several collision resolution techniques to manage the performance of a hash table.

It is a process of finding an alternate location. The collision resolution techniques can be named as-

Open Hashing (Separate Chaining)

Closed Hashing (Open Addressing)

Linear Probing

Quadratic Probing

Double Hashing

Linear Probing

Hashing in data structure results in an array index that is already occupied to store a value. In such a case, hashing performs a search operation and probes linearly for the next empty cell.

Linear probing in hash techniques is known to be the easiest way to resolve any collisions in hash tables. A sequential search can be performed to find any collision that occurred.

Linear Probing Example

Imagine you have been asked to store some items inside a hash table of size 30. The items are already sorted in a key-value pair format. The values given are: (3,21) (1,72) (63,36) (5,30) (11,44) (15,33) (18,12) (16,80) (46,99).

The $\text{hash}(n)$ is the index computed using a hash function and T is the table size. If slot index $= (\text{hash}(n) \% T)$ is full, then we look for the next slot index by adding 1 $((\text{hash}(n) + 1) \% T)$. If $(\text{hash}(n) + 1) \% T$ is also full, then we try $(\text{hash}(n) + 2) \% T$. If $(\text{hash}(n) + 2) \% T$ is also full, then we try $(\text{hash}(n) + 3) \% T$.

The hash table will look like the following:

Serial Number	Key	Hash	Array Index	Array Index after Linear Probing
1	3	$3\%30 = 3$	3	3
2	1	$1\%30 = 1$	1	1
3	63	$63\%30 = 3$	3	4
4	5	$5\%30 = 5$	5	5
5	11	$11\%30 = 11$	11	11
6	15	$15\%30 = 15$	15	15
7	18	$18\%30 = 18$	18	18
8	16	$16\%30 = 16$	16	16
9	46	$46\%30 = 16$	16	17

Double Hashing

The double hashing technique uses two hash functions. The second hash function comes into use when the first function causes a collision. It provides an offset index to store the value.

The formula for the double hashing technique is as follows:

$$(\text{firstHash}(\text{key}) + i * \text{secondHash}(\text{key})) \% \text{sizeOfTable}$$

Where i is the offset value. This offset value keeps incremented until it finds an empty slot.

For example, you have two hash functions: h_1 and h_2 . You must perform the following steps to find an empty slot:

Verify if $hash_1(key)$ is empty. If yes, then store the value on this slot.

If $hash_1(key)$ is not empty, then find another slot using $hash_2(key)$.

Verify if $hash_1(key) + hash_2(key)$ is empty. If yes, then store the value on this slot.

Keep incrementing the counter and repeat with $hash_1(key)+2hash_2(key)$, $hash_1(key)+3hash_2(key)$, and so on, until it finds an empty slot.

Double Hashing Example

Imagine you need to store some items inside a hash table of size 20. The values given are: (16, 8, 63, 9, 27, 37, 48, 5, 69, 34, 1).

$$h_1(n)=n\%20$$

$$h_2(n)=n\%13$$

$$h(n, i) = (h_1(n) + i h_2(n)) \bmod 20$$

n	$h(n,i) = (h'(n) + i2) \%20$
16	$i = 0, h(n,0) = 16$
8	$i = 0, h(n,0) = 8$
63	$i = 0, h(n,0) = 3$
9	$i = 0, h(n,0) = 9$
27	$i = 0, h(n,0) = 7$
37	$i = 0, h(n,0) = 17$

48	$l = 0, h(n,0) = 8$ $l = 0, h(n,1) = 9$ $l = 0, h(n,2) = 12$
5	$l = 0, h(n,0) = 5$

69	$l = 0, h(n,0) = 9$ $l = 0, h(n,1) = 10$
34	$l = 0, h(n,0) = 14$
1	$l = 0, h(n,0) = 1$

ALGORITHM:

- 1) Struct HashTable
Integer pointer to array
Integer variable to store size of array
- 2) Function getHashfn (key , struct HashTable)

Return key % size of array
- 3) Function Initialize (struct HashTable)
Iterate over the entire array and set each element to -1
- 4) Function PrintTable (struct HashTable)
Iterate a for loop from $i=0$ until $i < \text{size of array}$

If $\text{arr}[i]$ is not equal to -1 then Print $\text{arr}[i]$

Else Print NULL

5) Function Insertion (struct HashTable , int key , int index)

If arr[index] is equal to -1 then

Arr[index] ← key

Print the Hash Table with function PrintTable function

Return

Else

Set index=((index+1)%(h->size))

Call function Insertion recursively

6) Main function

Allocate memory for struct HashTable

Take input the size of array

Set size of Hash Table as the input taken

Allocate memory for array of Hash Table

Initialize all elements of Hash Table to -1 with Initialize function

Repeat until user does not enter the number of elements less than size of Hash Table

Take input the number of elements

Iterate from i=0 until i<number of elements

Take input the key

Calculate index with the help of Hash function

Insert the element with the help of function Insertion

PROBLEM SOLVING ON CONCEPT:

Linear Probing:

Size of table = h

Addr. of mapping = i

Start with at locⁿ where collision occ.
let it be i , then do sequential
search until,

$i, i+1, i+2, \dots, h, 1, 2, \dots, i-1$

- Desired index is empty
- If des. locⁿ not empty then if empty locⁿ is encountered
- It reaches locⁿ where search is.

Hash table is considered to be circular, hence technique is called closed hashing. Probe means key comparison.

Size of Hash Table

Size of Hash Table: 10

No. of elements: 8

Keys:

16, 66, 77, 31, 42, 52, 76, 67

Hash function: $H(K) = K \% \text{size}$

~~S-1(16)~~

Index:	S-1(16)	S-2(66)	S-3(77)	S-4(31)	S-5(42)
0					
1				31	31
2					42
3					
4					
5					
6	16	16	16	16	16
7		66 *	66	66	66
8			77 *	77	77
9					

Q-

S-2: Collision at index: 6

S-3: Collision at index: 7

Index	S-6 (52)	S-7 (76)	S-8 (67)	S-8 (67)
0			67	67*
1	31	31	67	31
2	42	42		42
3	52*	52		52
4				
5				
6	16	16		16
7	66	66		66
8	77	77		77
9		76*		76

S-6: Collision at Index: 2
 S-7: Collision at Index: 6, 7, 8
 S-8: Collision at Index: 7, 8, 9

CODE:

```

#include<stdio.h>
#include<string.h>
#include<stdlib.h>

struct HashTable
{
    int * arr;
    int size;
};

int getHashfn(int key , struct HashTable * h){
    return key % (h->size);
}

void Initialize(struct HashTable * h){
    for (int i = 0; i < h->size; i++)
    {
        h->arr[i]=-1;
    }
}
  
```

```

}

void PrintTable(struct HashTable * h){
    printf("-----\n");
    printf("| Index | Key |\n");
    printf("-----\n");
    for (int i = 0; i < h->size; i++)
    {
        if(h->arr[i]!=-1){
            printf("| %d | %2d |\n",i,h->arr[i]);
        }
        else{
            printf("| %d | NULL |\n",i);
        }
    }
    printf("-----\n");
}

void Insertion(struct HashTable * h, int key,int index){
    if(h->arr[index]==-1){
        h->arr[index]=key;
        PrintTable(h);
        return;
    }
    else{
        printf("Collision occurred at index:%d\n",index);
        index=((index+1)%(h->size));
        Insertion(h,key,index);
    }
}

int main()
{
    int size;
    struct HashTable * h=(struct HashTable *)malloc(sizeof(struct HashTable));

    printf("Enter the size of HashTable\n");
    scanf("%d",&size);

    h->size=size;
    h->arr=(int *)malloc(h->size * sizeof(int));

    Initialize(h);

    int num_ele=100000;

    while(num_ele>h->size){
        printf("Enter the number of elements you want to insert in Hash Table\n");
    }
}

```



```

scanf("%d",&num_ele);
if(num_ele>h->size){
    printf("Please enter the number of elements less the size of hash
table\n");
}
}

int ele;
for (int i = 0; i < num_ele; i++) {
    printf("Enter the key\n");
    scanf("%d",&ele);
    int index=getHashfn(ele,h);
    Insertion(h,ele,index);
}

return 0;
}

```

OUTPUT SCREENSHOT:

The screenshot shows the OnlineGDB web interface. The left sidebar contains navigation links: "Create New Project", "My Projects", "Classroom" (with a "new" badge), "Learn Programming", "Programming Questions", and "Logout". The main area displays the program's output:

```

input
Enter the size of HashTable
10
Enter the number of elements you want to insert in Hash Table
8
Enter the key
16
-----
| Index | Key |
-----
| 0     | NULL|
| 1     | NULL|
| 2     | NULL|
| 3     | NULL|
| 4     | NULL|
| 5     | NULL|
| 6     | 16  |
| 7     | NULL|
| 8     | NULL|
| 9     | NULL|
-----
Enter the key
66
Collision occurred at index:6

```

The bottom of the interface shows the Windows taskbar with various application icons and the system clock indicating 22:27 on 27-11-2022.

```
MySQL PROCEDURE - javatpoint | L-6.4: Linear Probing in Hashing with exa | GDB online Debugger | Compiler - C | +
onlinegdb.com
input
Enter the key
66
Collision occurred at index:6
-----
| Index | Key |
-----
| 0     | NULL |
| 1     | NULL |
| 2     | NULL |
| 3     | NULL |
| 4     | NULL |
| 5     | NULL |
| 6     | 16  |
| 7     | 66  |
| 8     | NULL |
| 9     | NULL |
-----
Enter the key
77
Collision occurred at index:7
-----
| Index | Key |
-----
```

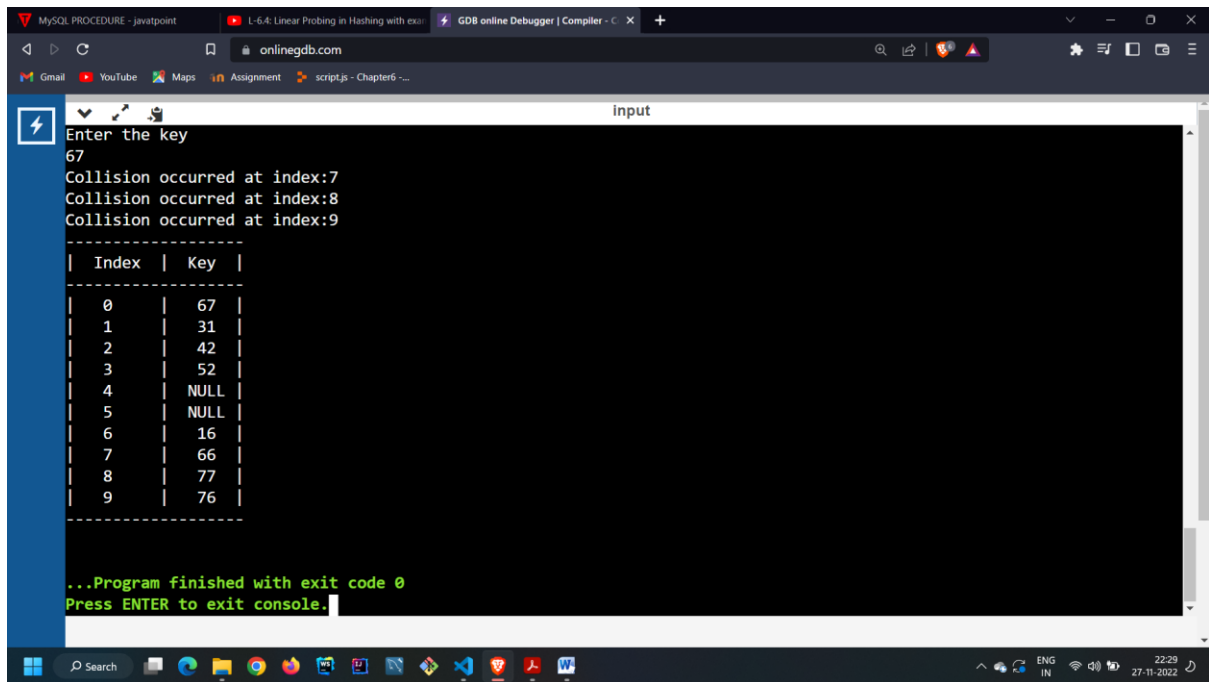
```
MySQL PROCEDURE - javatpoint | L-6.4: Linear Probing in Hashing with exa | GDB online Debugger | Compiler - C | +
onlinegdb.com
input
Enter the key
77
Collision occurred at index:7
-----
| Index | Key |
-----
| 0     | NULL |
| 1     | NULL |
| 2     | NULL |
| 3     | NULL |
| 4     | NULL |
| 5     | NULL |
| 6     | 16  |
| 7     | 66  |
| 8     | 77  |
| 9     | NULL |
-----
Enter the key
31
-----
| Index | Key |
-----
| 0     | NULL |
-----
```

```
MySQL PROCEDURE - javatpoint  L-6.4: Linear Probing in Hashing with exa...  GDB online Debugger | Compiler - C  X  +
onlinegdb.com
input
Enter the key
31
-----
| Index | Key |
-----
| 0     | NULL|
| 1     | 31  |
| 2     | NULL|
| 3     | NULL|
| 4     | NULL|
| 5     | NULL|
| 6     | 16  |
| 7     | 66  |
| 8     | 77  |
| 9     | NULL|
-----
Enter the key
42
-----
| Index | Key |
-----
| 0     | NULL|
| 1     | 31  |
```

```
MySQL PROCEDURE - javatpoint  L-6.4: Linear Probing in Hashing with exa...  GDB online Debugger | Compiler - C  X  +
onlinegdb.com
input
Enter the key
42
-----
| Index | Key |
-----
| 0     | NULL|
| 1     | 31  |
| 2     | 42  |
| 3     | NULL|
| 4     | NULL|
| 5     | NULL|
| 6     | 16  |
| 7     | 66  |
| 8     | 77  |
| 9     | NULL|
-----
Enter the key
52
Collision occurred at index:2
-----
| Index | Key |
-----
| 0     | NULL|
```

```
MySQL PROCEDURE - javatpoint | L-6.4: Linear Probing in Hashing with exa | GDB online Debugger | Compiler - C | X | +
onlinegdb.com
input
Enter the key
52
Collision occurred at index:2
-----
| Index | Key |
-----
| 0     | NULL|
| 1     | 31  |
| 2     | 42  |
| 3     | 52  |
| 4     | NULL|
| 5     | NULL|
| 6     | 16  |
| 7     | 66  |
| 8     | 77  |
| 9     | NULL|
-----
Enter the key
76
Collision occurred at index:6
Collision occurred at index:7
Collision occurred at index:8
-----
```

```
MySQL PROCEDURE - javatpoint | L-6.4: Linear Probing in Hashing with exa | GDB online Debugger | Compiler - C | X | +
onlinegdb.com
input
Enter the key
76
Collision occurred at index:6
Collision occurred at index:7
Collision occurred at index:8
-----
| Index | Key |
-----
| 0     | NULL|
| 1     | 31  |
| 2     | 42  |
| 3     | 52  |
| 4     | NULL|
| 5     | NULL|
| 6     | 16  |
| 7     | 66  |
| 8     | 77  |
| 9     | 76  |
-----
Enter the key
67
Collision occurred at index:7
Collision occurred at index:8
-----
```

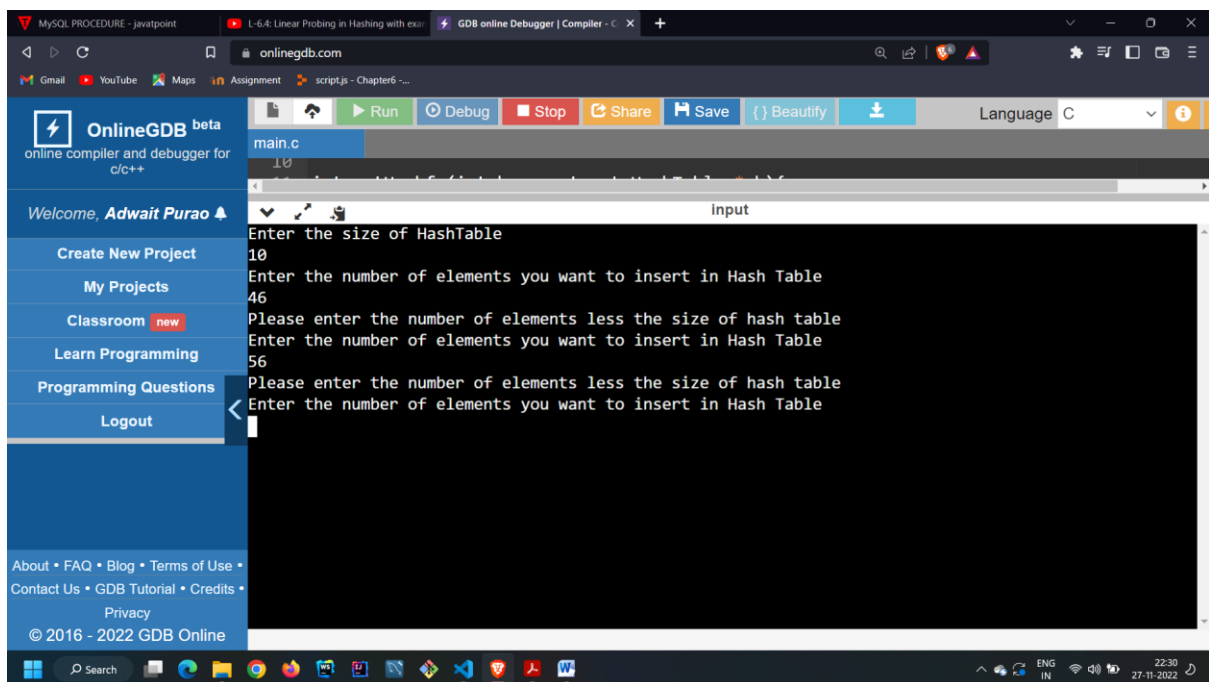


```
Enter the key
67
Collision occurred at index:7
Collision occurred at index:8
Collision occurred at index:9
-----
| Index | Key |
|-----|-----|
| 0     | 67  |
| 1     | 31  |
| 2     | 42  |
| 3     | 52  |
| 4     | NULL|
| 5     | NULL|
| 6     | 16  |
| 7     | 66  |
| 8     | 77  |
| 9     | 76  |
-----

...Program finished with exit code 0
Press ENTER to exit console.
```

Index	Key
0	67
1	31
2	42
3	52
4	NULL
5	NULL
6	16
7	66
8	77
9	76

Test Case : When we enter number of elements higher than size of Hash Table



```
main.c
10
Enter the size of HashTable
10
Enter the number of elements you want to insert in Hash Table
46
Please enter the number of elements less the size of hash table
Enter the number of elements you want to insert in Hash Table
56
Please enter the number of elements less the size of hash table
Enter the number of elements you want to insert in Hash Table
<
```

CONCLUSION:

In the following experiment we learnt about the concept of hashing . We learnt about various hash functions . We learnt the various methods to resolve collisions like Linear Probing , Quadratic Probing and Closed Hashing . We implemented the code to resolve the collisions using Linear Probing technique.