Name – Prerna Sunil Jadhav **SAP ID - 60004220127**

Experiment No – 10

AIM: Implementation Of Hashing Functions With Linear Probing Collision Resolution Techniques

Theory:

Hashing is a technique or process of mapping keys, and values into the hash table by using a hash function. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used.

Hashing Components:

- Hash Table: An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry. In simple terms, we can say that hash table is a generalization of array. Hash table gives the functionality in which a collection of data is stored in such a way that it is easy to find those items later if required. This makes searching of an element very efficient.
- Hash Function: A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in hash table. So, in simple terms we can say that a hash function is used to transform a given key into a specific slot index. Its main job is to map each and every possible key into a unique slot index. If every key is mapped into a unique slot index, then the hash function is known as a perfect hash function. It is very difficult to create a perfect hash function but our job as a programmer is to create such a hash function with the help of which the number of collisions are as few as possible. Collision is discussed ahead
- A good hash function should have following properties:
  - Efficiently computable.
  - Should uniformly distribute the keys (Each table position equally likely for each).
  - Should minimize collisions.
  - Should have a low load factor(number of items in table divided by size of the table).

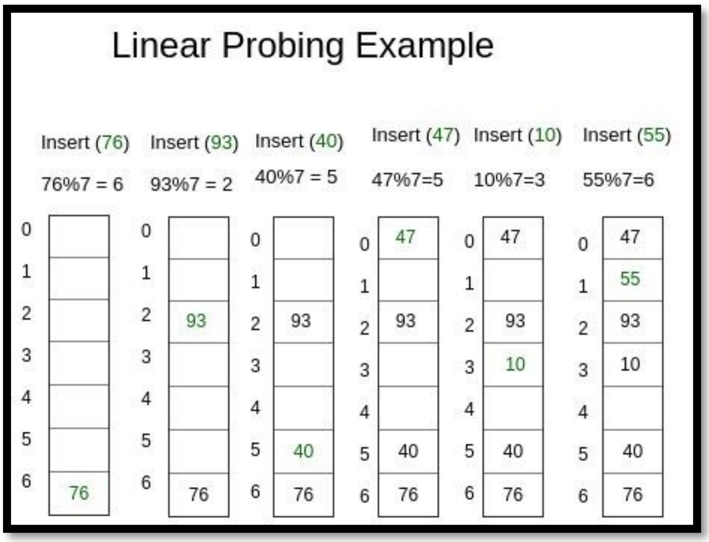
Algorithm:

```
Algorithm to insert a value in linear probing
Hashtable is an array of size = TABLE_SIZE
Step 1: Read the value to be inserted, key
Step 2:  let i = 0
Step 3: hkey = key% TABLE_SIZE
Step 4 :compute the index at which the key has to be inserted in hash table
                index = (hkey + i) % TABLE_SIZE
Step 5: if there is no element at that index then  insert the value at index and STOP
Step 6: If there is already an element at that index
                step 4.1: i = i+1
```

```
step 7:  if i < TABLE_SIZE then go to step 4

Algorithm to search a value in linear probing
Hashtable is an array of size = TABLE_SIZE
Step 1: Read the value to be searched, key
Step 2:  let i = 0
Step 3: hkey = key% TABLE_SIZE
Step 4: compute the index at which the key can be found
               index = (hkey+ i) % TABLE_SIZE
Step 5: if the  element at that index is same as the search value then print element found
and  STOP
Step 6: else
                step 4.1: i = i+1
step 7:  if i < TABLE_SIZE then go to step 4
```

Example:



Program:

```
#include <stdio.h>
#include <stdlib.h>
#define TABLE_SIZE 10
int h[TABLE_SIZE] = {NULL};
void insert()
{
    int key, index, i, flag = 0, hkey;
    printf("\nenter a value to insert into hash table\n");
    scanf("%d", &key);
    hkey = key % TABLE_SIZE;
    for (i = 0; i < TABLE_SIZE; i++)
```

```
    {

        index = (hkey + i) % TABLE_SIZE;

        if (h[index] == NULL)
        {
            h[index] = key;
            break;
        }
    }
    if (i == TABLE_SIZE)

        printf("\nelement cannot be inserted\n");
}
void search()
{
    int key, index, i, flag = 0, hkey;
    printf("\nenter search element\n");
    scanf("%d", &key);
    hkey = key % TABLE_SIZE;
    for (i = 0; i < TABLE_SIZE; i++)
    {
        index = (hkey + i) % TABLE_SIZE;
        if (h[index] == key)
        {
            printf("value is found at index %d", index);
            break;
        }
    }
    if (i == TABLE_SIZE)
        printf("\n value is not found\n");
}
void display()
{
    int i;
    printf("\nelements in the hash table are \n");
    for (i = 0; i < TABLE_SIZE; i++)
        printf("\nat index %d \t value =  %d", i, h[i]);
}
main()
{
    printf("Prerna Sunil Jadhav - 60004220127\n");

    int opt, i;
    while (1)
    {
```

```
        printf("\nPress 1. Insert\t 2. Display \t3. Search \t4.Exit \n");
        scanf("%d", &opt);
        switch (opt)
        {
        case 1:
            insert();
            break;
        case 2:
            display();
            break;
        case 3:
            search();
            break;
        case 4:
            exit(0);
        }
    }
}
```

OUTPUT:

```
Prerna Sunil Jadhav - 60004220127

Press 1. Insert   2. Display      3. Search       4.Exit
1

enter a value to insert into hash table
12

Press 1. Insert   2. Display      3. Search       4.Exit
1

enter search element
32

 value is not found

Press 1. Insert   2. Display      3. Search       4.Exit
1

enter a value to insert into hash table
12

Press 1. Insert   2. Display      3. Search       4.Exit
2
```

```
elements in the hash table are

at index 0       value =  0
at index 1       value =  0
at index 2       value =  12
at index 3       value =  12
at index 4       value =  0
at index 5       value =  45
at index 6       value =  0
at index 7       value =  0
at index 8       value =  0
at index 9       value =  0
```

Conclusion:

- In linear probing,
  - When collision occurs, we linearly probe for the next bucket.
  - We keep probing until an empty bucket is found.
  - Advantage-
    - It is easy to compute.
  - Disadvantage-
    - The main problem with linear probing is clustering.
    - Many consecutive elements form groups.
    - Then, it takes time to search an element or to find an empty bucket.
  - Time Complexity-
    - Worst time to search an element in linear probing is O (table size).
      - This is because-
        - Even if there is only one element present and all other elements are deleted.
        - Then, “deleted” markers present in the hash table makes search the entire table.