K. J. Somaiya College of Engineering, Mumbai-77
(Autonomous College Affiliated to University of Mumbai)

Batch: B1 **Roll No.: 1711072**

Experiment No. 9

Grade: AA / AB / BB / BC / CC / CD /DD

Title: Hashing and Collision Resolution Techniques

Objective: Implementation of Hashing using collision resolution methods like linear and quadratic probing.

Expected Outcome of Experiment:

CO	Outcome
CO3	Demonstrate sorting and searching methods

Books/ Journals/ Websites referred:

Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein; (CLRS) "Introduction to Algorithms", Third Edition, The MIT Press.

Abstract:

NODE STRUCTURE:

1. char name[30] //to store the names with max size 30.
2. int age //to store age of users.
3. int key //to store unique key value of each user.

OPERATIONS:

1. **int hash(int key)**

   Used for generating the hash value for the key value passed.

2. **void insert()**

   Used for inserting entries into the hash table.

3. **int search(int key)**

   Used for searching for a user with his key value.

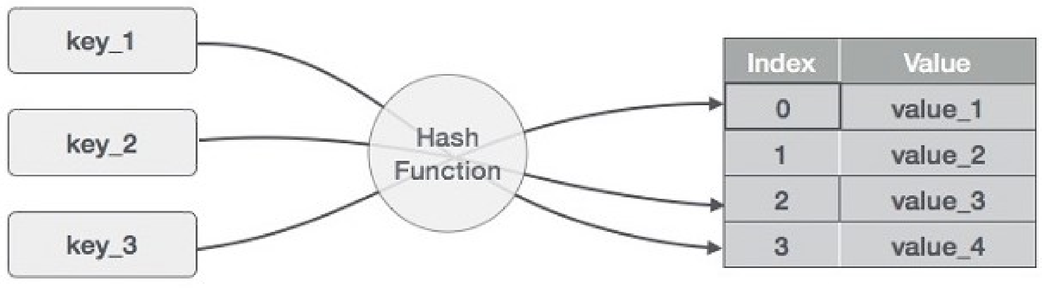
4. **void display()**

   Used for displaying the table of entries in an aligned form.

Related Theory: -

Hashing:

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key, value) format.



Hash function:

The problem at hands is to speed up searching. Consider the problem of searching an array for a given value. If the array is not sorted, the search might require examining each and all elements of the array. If the array is sorted, we can use the binary search, and therefore reduce the worse-case runtime complexity to O(log n). We could search even faster if we know in advance the index at which that value is located in the array.

Suppose we do have that magic function that would tell us the index for a given value. With this magic function our search is reduced to just one probe, giving us a constant runtime O(1). Such a function is called a **hash function**. A hash function is a function which when given a key, generates an address in the table. A hash function that returns a unique hash number is called a **universal hash function**. In practice it is extremely hard to assign unique numbers to objects. The latter is always possible only if you know (or approximate) the number of objects to be processed.

Hash Table:

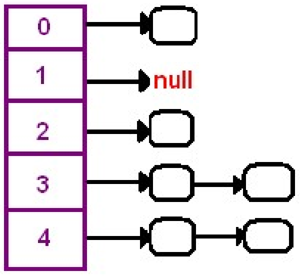
Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

Collisions:

When we put objects into a hashtable, it is possible that different objects (by the *equals()* method) might have the same hashcode. This is called a **collision**. Here is the example of collision. Two different strings ""Aa" and "BB" have the same key:

```
"Aa" = 'A' * 31 + 'a' = 2112
"BB" = 'B' * 31 + 'B' = 2112
```



How to resolve collisions? Where do we put the second and subsequent values that hash to this same location? There are several approaches in dealing with collisions. One of them is based on idea of putting the keys that collide in a linked list! A hash table then is an array of lists!! This technique is called a *separate chaining* collision resolution.

The big attraction of using a hash table is a constant-time performance for the basic operations `add, remove, contains, size`. Though, because of collisions, we cannot guarantee the constant runtime in the worst-case. Why? Imagine that all our objects collide into the same index. Then searching for one of them will be equivalent to searching in a list that takes a liner runtime. However, we can guarantee an expected constant runtime, if we make sure that our lists won't become too long. This is usually implemented by maintaining a *load factor* that keeps a track of the average length of lists. If a load factor approaches a set in advanced threshold, we create a bigger array and *rehash* all elements from the old table into the new one.

Another technique of collision resolution is a *linear probing*. If we cannot insert at index k, we try the next slot k+1. If that one is occupied, we go to k+2, and so on. This is quite simple approach but it requires new thinking about hash tables. You need to stop iterating once you complete searching all indexes in worst case scenario.

Implementation Details:

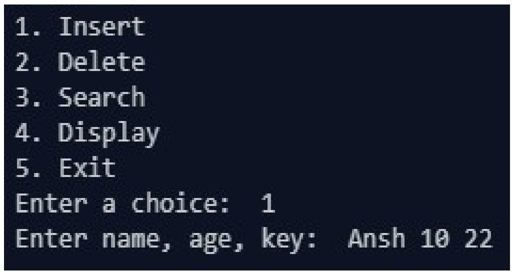
```
#include <stdio.h>
#include <stdlib.h>

int hash(int key){
  return key%20;
}
typedef struct node{
  char name[30];
  int age;
  int key;
}node;
node* table[20]={NULL};

void insert(){
  int i, iterations=1;
  node* new_data;
  new_data=(node*)malloc(sizeof(node));
  printf("Enter name, age, key: ");
  scanf("%s %d %d", new_data->name,&new_data->age, &new_data-
>key);
```

```
  i=hash(new_data->key);
  while(table[i]!=NULL && iterations<=20){
      i=(i+1)%20;
      iterations++;
    }
    if(iterations>=20)
      printf("Cannot enter more data.");
    else
      table[i]=new_data;
}

int search(int key){
  int i;
  int search_hash=hash(key);
  int iterations=0;
  while((table[search_hash]==NULL || table[search_hash]-
>key!=key) && iterations<=20){
    search_hash=(search_hash+1)%20;
    iterations++;
  }
  if(iterations>=20)
    return -1;
  else
    return search_hash;
  iterations=1;
}

void display(){
  printf("Index\tKey\tName\tAge\n");
  for(int i=0;i<20;i++){
    if(table[i]==NULL)
      continue;
    else{
      printf("%d\t%d\t%s\t%d\n",i, table[i]->key, table[i]-
>name, table[i]->age);
    }
  }
}

void main(){
  int choice, key;
  do{
    printf("\n1. Insert\n2. Delete\n3. Search\n4. Display\n5.
Exit\nEnter a choice: ");
```

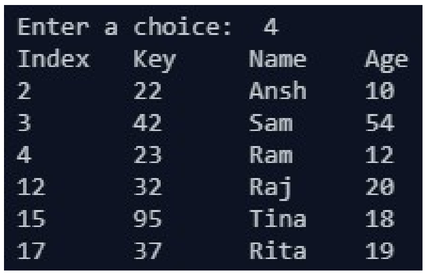
```
    scanf("%d", &choice);
    switch(choice){
      case 1:
      insert();
      break;
      case 2:
      printf("Enter key to delete: ");
      scanf("%d", &key);
      int val=search(key);
      if(val!=-1){
      printf("Person found at index: %d", val);
      table[val]=NULL;
      printf("\nDeleted data.");
      }
      else
        printf("Person not found");
      break;
      case 3:
      printf("Enter key of person to search: ");
      scanf("%d", &key);
      int value=search(key);
      if(value!=-1)
      printf("Person found at index: %d\nKey: %d \tName: %s
\tAge: %d", value,table[value]->key,table[value]->name,
table[value]->age);
      else
        printf("Person not found");
      break;
      case 4:
      display();
      break;
      case 5: exit(1);
    }
  }while(choice!=5);
}
```

For verification, my code is available on:
https://repl.it/@ARGHYADEEPDAS/HashingInC

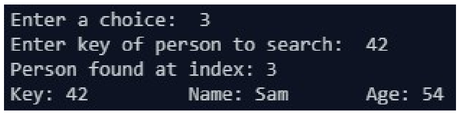
OUTPUT SCREEN:



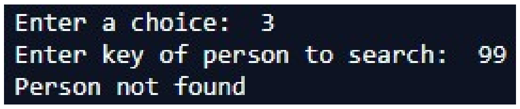
```
1. Insert
2. Delete
3. Search
4. Display
5. Exit
Enter a choice:  1
Enter name, age, key:  Ansh 10 22
```



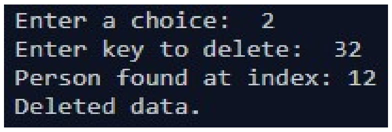
```
Enter a choice:  4
Index    Key      Name     Age
2        22       Ansh     10
3        42       Sam      54
4        23       Ram      12
12       32       Raj      20
15       95       Tina     18
17       37       Rita     19
```



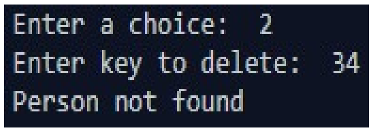
```
Enter a choice:  3
Enter key of person to search:  42
Person found at index: 3
Key: 42          Name: Sam         Age: 54
```



```
Enter a choice:  3
Enter key of person to search:  99
Person not found
```



```
Enter a choice:  2
Enter key to delete:  32
Person found at index: 12
Deleted data.
```



```
Enter a choice:  2
Enter key to delete:  34
Person not found
```

CONCLUSION:

The program ran successfully as we were able to implement menu driven program for hashing and were able to resolve collisions using linear and quadratic probing successfully.