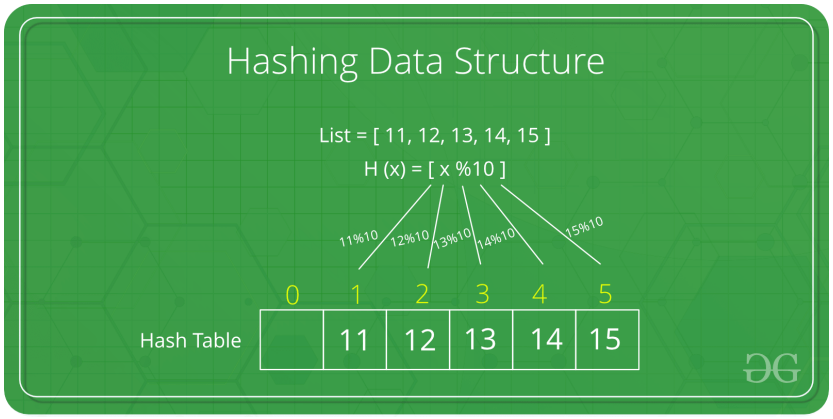
Name	Maureen Miranda
UID no.	2022300060
Experiment No.	9

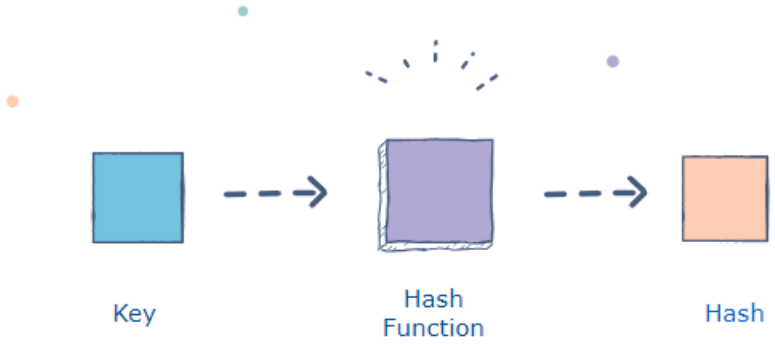
AIM:	Implement Hashing
Program 1	
PROBLEM STATEMENT :	Implement Hashing using Separate chaining
THEORY:	Hashing is a technique or process of mapping keys, and values into the hash table by using a hash function. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used. Let a hash function H(x) maps the value x at the index x%10 in an Array. For example if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.  For example, in hash tables, developers store data -- perhaps a customer record -- in the form of key and value pairs. The key helps identify the data and operates as an input to

the hashing function, while the hash code or the integer is then mapped to a fixed size.

Hash tables support functions that include the following:

insert (key, value)
get (key)
delete (key)

Hashing is the process of converting a given key into another value. A hash function is used to generate the new value according to a mathematical algorithm. The result of a hash function is known as a hash value or simply, a hash.



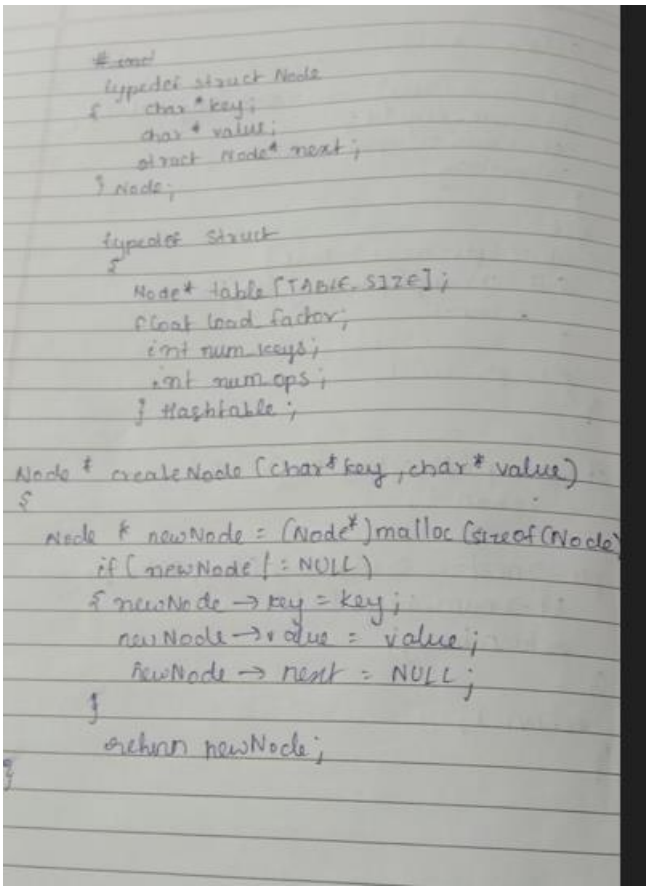
Algorithm :
Algorithm: Hash Table Implementation

Data Structures:
- Define a structure for a linked list node (Node) to store key-value pairs and a pointer to the next node.
- Define a structure for the HashTable that contains an array of linked list nodes, load factor, the number of keys present, the number of occupied indices in the array, and the number of operations done so far.

Functions:
1. createNode(char *key, char *value)
   - Allocate memory for a new Node.
   - Set the key and value for the new Node.
   - Initialize the next pointer to NULL.

	- Return the new Node. 2. createHashTable() - Allocate memory for a new HashTable. - Initialize the array of linked list nodes to NULL. - Initialize the load factor, number of keys, number of occupied indices, and number of operations to 0. - Return the new HashTable. 3. key_to_int(char *key) - Calculate a hash value for a given key using the sum of ASCII values of its characters. - Return the hash value. 4. insert_key_value(HashTable *ht, char *key, char *value) - Calculate the hash value for the key using key_to_int function. - Find the corresponding index (hkey) in the HashTable. - Increment the number of operations. - If the index is empty (NULL), create a new Node and add it as the head of the linked list. - Handle duplicates by checking for matching keys and return an error. - Update the number of keys, occupied indices, and return the index (hkey). 5. search_key(HashTable *ht, char *key) - Calculate the hash value for the key using key_to_int function. - Find the corresponding index (hkey) in the HashTable. - Increment the number of operations. - Search for the key in the linked list at the index (hkey). - If found, return the index; otherwise, return an error. 6. delete_key(HashTable *ht, char *key) - Calculate the hash value for the key using key_to_int function. - Find the corresponding index (hkey) in the HashTable. - Increment the number of operations. - If the key is at the head of the linked list at the index (hkey), remove it. - If the key is in the middle of the linked list, remove it and update pointers. - If the key is at the end of the linked list, remove it. - If the key is not found, return an error. - Update the number of keys. - Return the index (hkey).

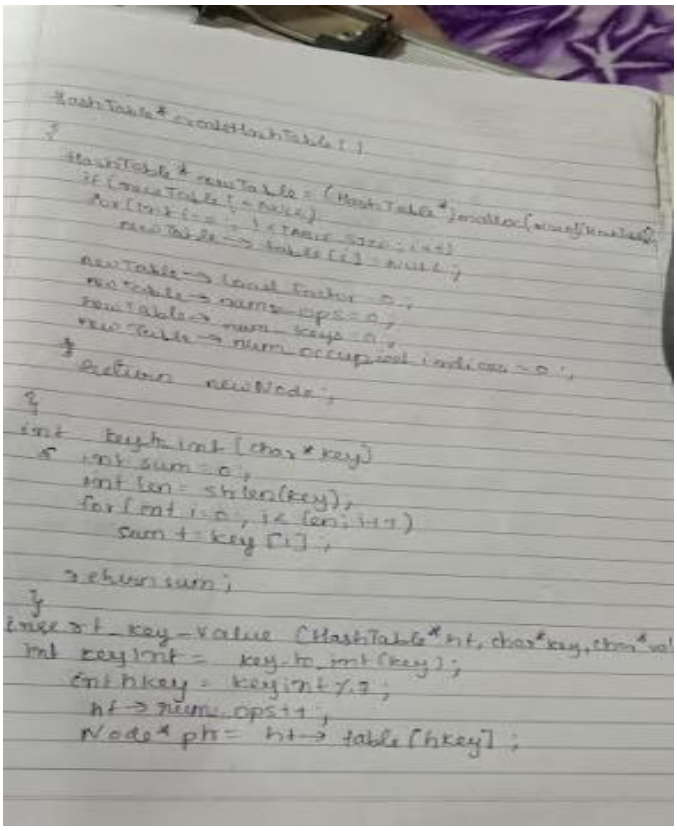
	7. display(HashTable *ht) - Display the contents of the HashTable, including occupied indices and linked list nodes. - Print "NULL" if a linked list is empty. - Increment the number of operations. 8. get_load_factor(HashTable *ht) - Calculate and return the load factor (number of keys/size of the array). - Increment the number of operations. 9. get_avg_probes(HashTable *ht) - Calculate and return the average probes (number of operations/number of keys). - Increment the number of operations. Main Function: - Initialize a HashTable using createHashTable. - Perform insert, search, and delete operations on the HashTable. - Display the HashTable's contents and calculate load factor and average probes. End of Algorithm.

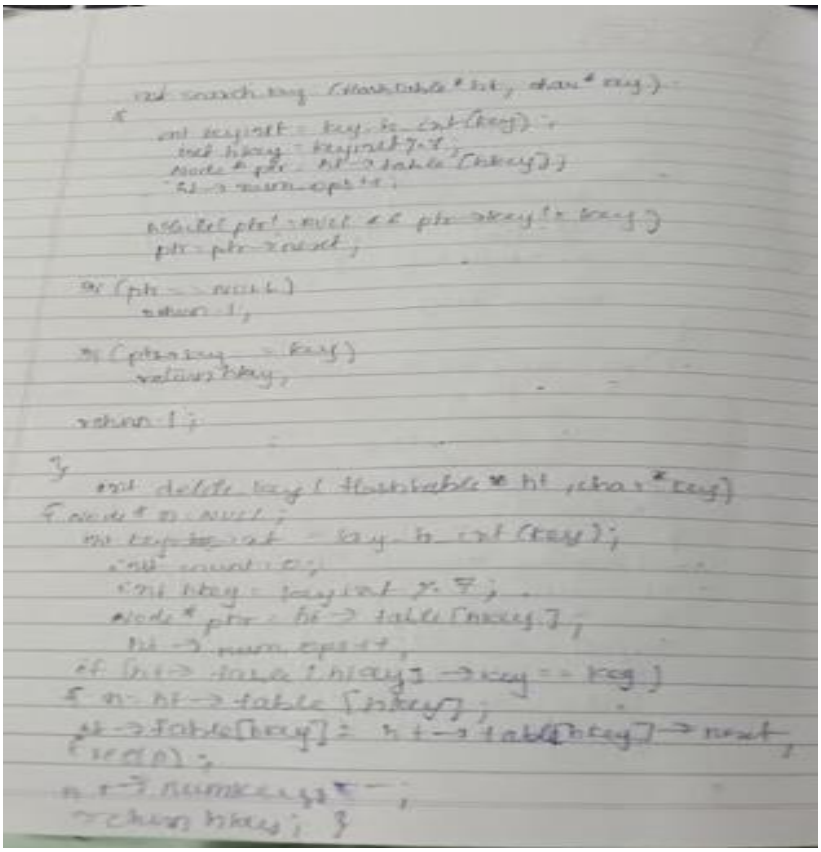


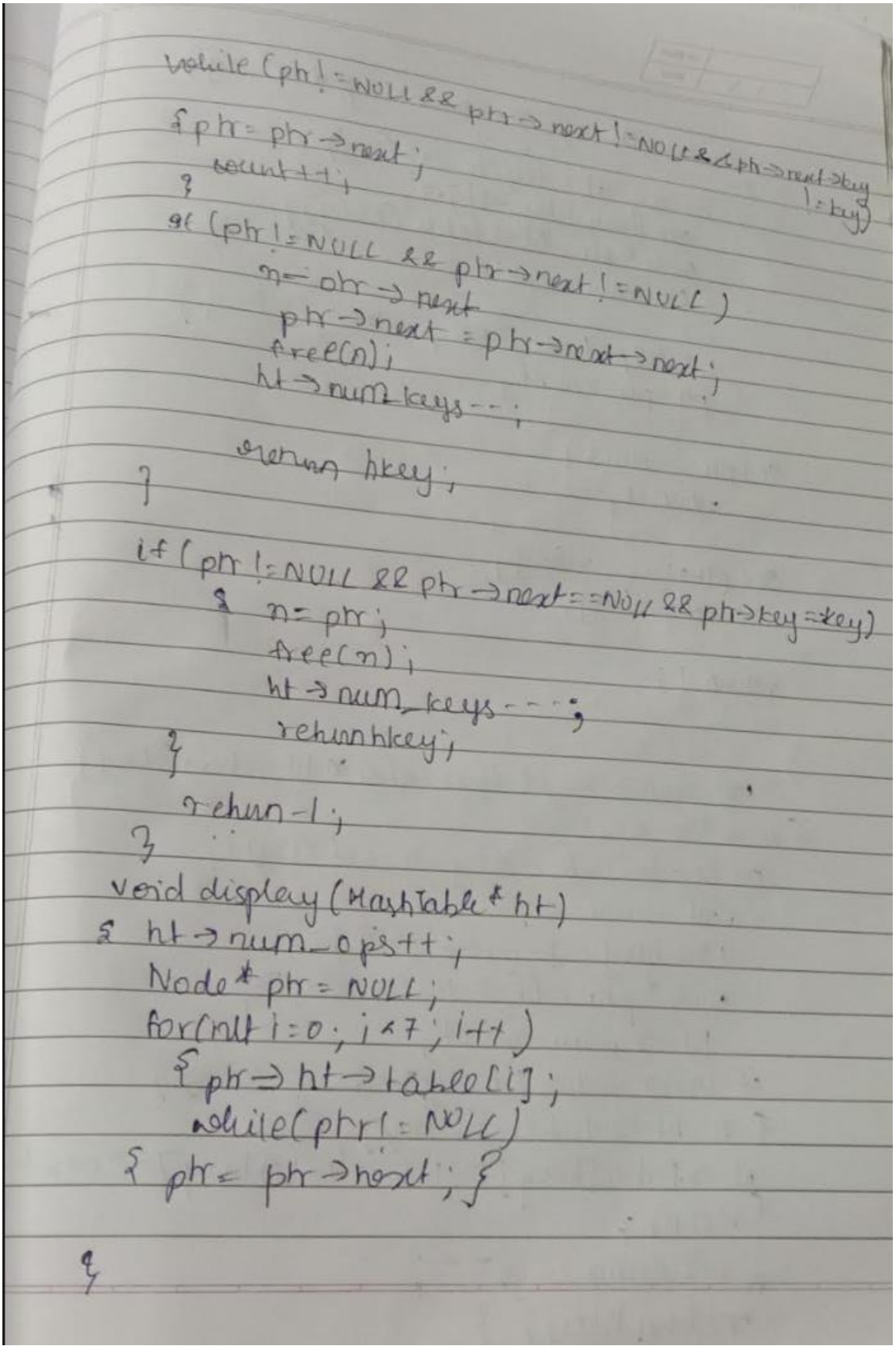
```
#endif
typedef struct Node
{   char *key;
    char *value;
    struct Node* next;
} Node;

typedef struct
{
    Node* table[TABLE_SIZE];
    float load_factor;
    int num_keys;
    int num_ops;
} HashTable;

Node * createNode(char* key, char* value)
{
  Node * newNode = (Node*)malloc(sizeof(Node));
    if(newNode != NULL)
    { newNode -> key = key;
      newNode -> value = value;
      newNode -> next = NULL;
    }
    return newNode;
}
```





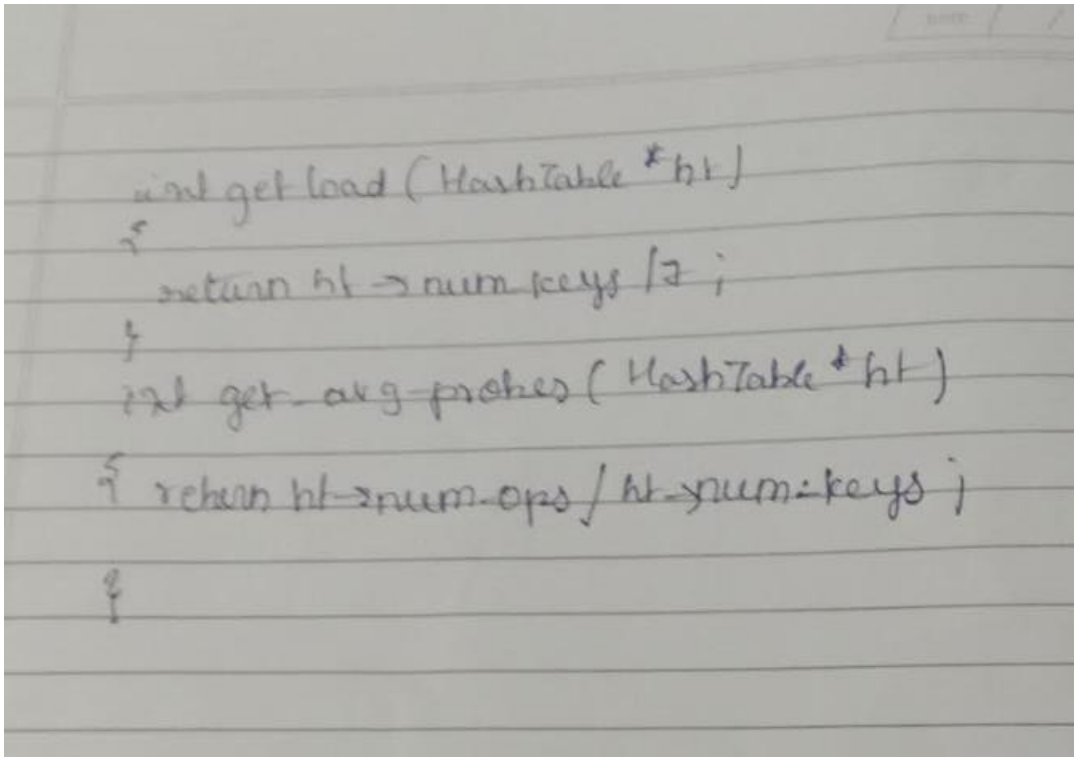


```
while (ptr != NULL && ptr->next != NULL && ptr->next->key
                                                != key)
{ ptr = ptr->next;
  count++;
}
if (ptr != NULL && ptr->next != NULL)
    n = ptr->next
    ptr->next = ptr->next->next;
    free(n);
    ht->num_keys--;

    return hkey;
}

if (ptr != NULL && ptr->next == NULL && ptr->key == key)
  { n = ptr;
    free(n);
    ht->num_keys--;
    return hkey;
  }
  return -1;
}
void display (HashTable * ht)
{ ht->num_ops++;
  Node * ptr = NULL;
  for(int i=0; i<7; i++)
    { ptr = ht->table[i];
      while(ptr != NULL)
  { ptr = ptr->next; }

}
```



```
int get_load (HashTable *ht)
{
  return ht->num_keys /7 ;
}
int get_avg_probes (HashTable *ht)
{ return ht->num_ops / ht->num_keys ;
}
```

PROGRAM:	

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

#define TABLE_SIZE 7

typedef struct Node {
   char *key;
   char *value;
   struct Node* next;
} Node;

typedef struct {
   Node *table[TABLE_SIZE];
   float load_factor;
   // num of keys present
   int num_keys;
   // num of array indices of the table that are occupied
   int num_occupied_indices;
   // num of ops done so far
   int num_ops;
} HashTable;


Node *createNode(char *key, char *value) {
   Node* newNode = (Node*)malloc(sizeof(Node));
   if (newNode != NULL) {
      newNode->key = key;
      newNode->value = value;
      newNode->next = NULL;
   }
   return newNode;
}

HashTable* createHashTable() {
```

```
    HashTable* newTable = (HashTable*)malloc(sizeof(HashTable));
    if (newTable != NULL) {
        for (int i = 0; i < TABLE_SIZE; i++) {
            newTable->table[i] = NULL;
        }


        newTable->load_factor=0;
        newTable->num_ops=0;
        newTable->num_keys=0;
        newTable->num_occupied_indices=0;


    }
    return newTable;
}


// use sum of ascii values to convert string to int
int key_to_int(char* key){
    int sum=0;
int len = strlen(key);
for(int i=0;i<len;i++)
{
sum +=key[i];
}
return sum;
}

// return the index position in the table where the insertion happens
// return -1 if insertion fails
int insert_key_value(HashTable *ht, char* key, char* value)
{
int keyint = key_to_int(key);
int hkey = keyint%7;
ht->num_ops++;
Node*ptr=ht->table[hkey];
if(ptr==NULL)
{
```

```
ht->table[hkey]=createNode(key,value);
ht->num_keys++;

ht->num_occupied_indices++;
return hkey;
}
else
{

while(ptr->next!=NULL)
{
if(strcmp(ptr->key,key)==0)
{
   printf("\nERROR duplicate key not allowed\n\n\n");
   return -1;
}
ptr= ptr->next;
}
if(strcmp(ptr->key,key)==0)
{
   printf("\nERROR duplicate key not allowed\n\n\n");
   return -1;
}
ptr->next=createNode(key,value);
ht->num_keys++;

return hkey;
}
return -1;
}



int search_key(HashTable *ht, char* key)
{
int keyint = key_to_int(key);
int hkey = keyint%7;
Node * ptr=ht->table[hkey];
ht->num_ops++;
while(ptr!=NULL&& ptr->key!=key)
{
```

```
ptr=ptr->next;
}
if(ptr==NULL)
{

printf("\n\nELEMENT NOT FOUND\n\n");
return -1;}
if(ptr->key==key)
{
return hkey;
}
return -1;
}




int delete_key(HashTable *ht, char* key)
{
Node* n = NULL;
int keyint = key_to_int(key);
int count=0;
int hkey = keyint%7;
Node*ptr=ht->table[hkey];
ht->num_ops++;
if(ht->table[hkey]->key==key)
{
 n= ht->table[hkey];
ht->table[hkey]=ht->table[hkey]->next;
free(n);
ht->num_keys--;
return hkey;
}
while(ptr!=NULL&& ptr->next!=NULL&&ptr->next->key!=key)
{
ptr=ptr->next;
count++;
}
if(ptr!=NULL&&ptr->next!=NULL)
```

```
{
n=ptr->next;
ptr->next=ptr->next->next;
free(n);
ht->num_keys--;

return hkey;
}
if(ptr!=NULL&&ptr->next==NULL&&ptr->key==key)
{
  n=ptr;
  free(n);
  ht->num_keys--;
  return hkey;
}
printf("\n\nerror : key %s not present\n\n",key);
return -1;
}


void display(HashTable *ht)
{
   ht->num_ops++;
   Node* ptr=NULL;
  printf("\n\n");

   for(int i=0;i<7;i++)
      {
         printf("%d ",i);
         ptr=ht->table[i];
         while(ptr!=NULL)
         {printf("||%s - %s|| --> ",ptr->key,ptr->value);
          ptr=ptr->next;
         }
         printf("NULL");
         printf("\n");
      }


}
```

```
// this equals the number of keys in table/size of table
int get_load_factor(HashTable *ht)
{
   printf("\nload factor=%d/%d\nvalue=",ht->num_keys,7);
   return ht->num_keys/7;
}

// this equals the number of operations done so far/num of elems in table
int get_avg_probes(HashTable *ht)
{
   printf("\navg probes=");
   printf("%d / %d \nvalue=", ht->num_ops,ht->num_keys);
   return ht->num_ops/ht->num_keys;
}




int main() {
   // Sample code to create and use the hash table.
   HashTable *ht = createHashTable();
/*// -> Insert the following key, values in the table:
// 1. 'first name' -> <your first name>
// 2. 'last name' -> <your last name>
// 3. 'uid' -> <your uid>
// 4. 'sport' -> <your favorite sport>
// 5. 'food' -> <your favorite food>
// 6. 'holiday' -> <your favorite holiday destination>
// 7. 'role_model' -> <your role model>
// 8. 'subject' -> <your favourite subject>
// 9. 'song' -> <your favourite song>
// 10. 'movie' -> <your favorite movie>
// 11. 'colour' -> <your favorite colour>
// 12. 'book' -> <your favorite book>*/
   insert_key_value(ht, "firstname", "maureen");//1
   insert_key_value(ht, "lastname", "miranda");//2
   insert_key_value(ht, "uid", "2022300060");
   insert_key_value(ht, "sport", "badminton");
   insert_key_value(ht, "food", "noodles");
  insert_key_value(ht, "holiday", "Hawaii");
```

```
  insert_key_value(ht, "role_model", "TaylorSwift");
  insert_key_value(ht, "subject", "DataStructures");
  insert_key_value(ht, "song", "BadBlood");
  insert_key_value(ht, "movie", "Divergent");
  insert_key_value(ht, "colour", "Red");
  insert_key_value(ht, "book", "TheSubtleArtOfNotGivingA****");
display(ht);
//deletion displayed
delete_key(ht,"subject");
display(ht);
//search
printf("\nhashindex of rolemodel = %d",search_key(ht,"role_model"));
printf("\n\n");
insert_key_value(ht, "song", "DriversLicense");
delete_key(ht,"rainbow");
printf("index=%d\n\n",search_key(ht,"bird"));
printf("%d\n ",get_load_factor(ht));
printf("%d ",get_avg_probes(ht));
return 0;
}
```

RESULT:

```
int main() {
    // Sample code to create and use the hash table.
    HashTable *ht = createHashTable();

    insert_key_value(ht, "firstname", "maureen");//1
    insert_key_value(ht, "lastname", "miranda");//2
    insert_key_value(ht, "uid", "2022300060");
    insert_key_value(ht, "sport", "badminton");
    insert_key_value(ht, "food", "noodles");
   insert_key_value(ht, "holiday", "Hawaii");
   insert_key_value(ht, "role_model", "TaylorSwift");
   insert_key_value(ht, "subject", "DataStructures");
   insert_key_value(ht, "song", "BadBlood");
   insert_key_value(ht, "movie", "Divergent");
   insert_key_value(ht, "colour", "Red");
   insert_key_value(ht, "book", "TheSubtleArtOfNotGivingA****");
display(ht);
//deletion displayed
delete_key(ht,"subject");
display(ht);
//search
printf("\nhashindex of rolemodel = %d",search_key(ht,"role_model"));
printf("\n\n");
insert_key_value(ht, "song", "DriversLicense");
delete_key(ht,"rainbow");
printf("index=%d\n\n",search_key(ht,"bird"));
printf("%d\n ",get_load_factor(ht));
printf("%d ",get_avg_probes(ht));
return 0;
}
```

```
0 ||uid - 2022300060|| --> ||book - TheSubtleArtOfNotGivingA****|| --> NULL
1 ||sport - badminton|| --> ||role_model - TaylorSwift|| --> NULL
2 ||colour - Red|| --> NULL
3 ||firstname - maureen|| --> ||subject - DataStructures|| --> NULL
4 ||food - noodles|| --> ||holiday - Hawaii|| --> NULL
5 ||song - BadBlood|| --> ||movie - Divergent|| --> NULL
6 ||lastname - miranda|| --> NULL


0 ||uid - 2022300060|| --> ||book - TheSubtleArtOfNotGivingA****|| --> NULL
1 ||sport - badminton|| --> ||role_model - TaylorSwift|| --> NULL
2 ||colour - Red|| --> NULL
3 ||firstname - maureen|| --> NULL
4 ||food - noodles|| --> ||holiday - Hawaii|| --> NULL
5 ||song - BadBlood|| --> ||movie - Divergent|| --> NULL
6 ||lastname - miranda|| --> NULL

hashindex of rolemodel = 1


ERROR duplicate key not allowed




error : key rainbow not present


ELEMENT NOT FOUND

index=-1
```

```
error : key rainbow not present


ELEMENT NOT FOUND

index=-1


load factor=11/7
value=1

avg probes=19 / 11
value=1
```

Conclusion : In this experiment, we implemented a basic hash table data structure. We learned the importance of a good hash function to evenly distribute key-value pairs in the table and the need for collision handling mechanisms to manage situations where multiple keys map to the same index. Our code provides methods to insert, search, and delete key-value pairs while maintaining efficient performance.

Understanding hash tables is fundamental in computer science and is widely used in real-world applications, such as databases, caches, and more, to provide fast and efficient data access. This experiment has given us practical insights into the inner workings of hash tables and their

associated operations, which are valuable for further study in data structures and algorithm design.