|  |  |
|---|---|
| **Name** | Shubhan Singh |
| **UID no.** | 2022300118 |
| **Experiment No.** | 9 |

| **Program 1** |  |
|---|---|
| **PROBLEM STATEMENT :** | *Implement Hashing using collision resolution strategy of Double Hashing* |
| **ALGORITHM:** | **- ‘createKeyValue' Function:**<br>1. Create a new KeyValue structure.<br>2. Allocate memory for the key and value strings.<br>3. Copy the input key and value strings to the allocated memory.<br>4. Set the isDeleted field to false.<br>5. Return the created KeyValue structure.<br><br>**-‘createHashTable’ Function:**<br>1. Create a new HashTable structure.<br>2. Allocate memory for the array of KeyValue pointers.<br>3. Initialize each element of the array to NULL.<br>4. Set size to TABLE_SIZE, load_factor to 0, num_keys to 0, num_occupied_indices to 0, and num_ops to 0.<br>5. Return the created HashTable structure.<br><br>**-‘key_to_int’ Function:**<br>1. Initialize hash to 0 and ind to 0.<br>2. Loop through each character of the key until the null terminator is reached: |

```
    a. Add the ASCII value of the character plus 128 to the hash.
    b. Increment ind.
3. Take the modulo of hash with TABLE_SIZE.
4. Return the resulting hash.
```

-‘**secondhash’ Function:**

```
1. Calculate the second hash using the formula: 11 - (x % 11).
2. Return the calculated second hash.
```

-‘**insert_key_value’ Function Algorithm:**

```
1. Calculate the first hash using key_to_int function.
2. Create a new KeyValue structure with the provided key and value.
3. If the slot at the first hash is empty:
    a. Insert the new KeyValue structure at that slot.
4. Else, if the slot at the first hash is marked as deleted:
    a. Update the existing KeyValue structure with the new key and value.
    b. Mark the KeyValue structure as not deleted.
5. Else, handle collision using double hashing:
    a. Calculate the second hash using secondhash function.
    b. Start probing with an initial index equal to the first hash.
    c. Loop until an empty slot or a deleted slot is found:
        i. If a deleted slot is found:
            - Update the existing KeyValue structure with the new key and value.
            - Mark the KeyValue structure as not deleted.
            - Break the loop.
        ii. Increment the index by the second hash and take the modulo of TABLE_SIZE.
        iii. If the index becomes equal to the initial first hash, return 1 to indicate a potential loop.
    d. If the loop completes without finding an empty or deleted slot:
        i. Insert the new KeyValue structure at the calculated index.
6. Update the counters for the number of keys, occupied indices, and operations.
7. Return the index where the insertion happened.
```

-‘**search_key’ Function:**

```
1. Calculate the first hash using key_to_int function.
2. Set the initial index to the first hash.
3. If the slot at the first hash is empty, return NULL (key not found).
4. If the slot at the first hash matches the key and is not marked as deleted, return its value.
5. Else, handle collision using double hashing:
    a. Calculate the second hash using secondhash function.
    b. Start probing with an initial index equal to the first hash.
    c. Loop until an empty slot or a deleted slot is found:
        i. If the slot matches the key and is not marked as deleted, return its value.
        ii. Increment the index by the second hash and take the modulo of TABLE_SIZE.
        iii. If the index becomes equal to the initial first hash, return NULL (key not found).
6. If the loop completes without finding an empty or deleted slot, return NULL (key not found).
```

-‘**delete_key’ Function:**

```
1. Save the current value of num_ops in a variable n.
2. Call search_key to check if the key exists in the table.
3. If the key is not found, return -1 (deletion failed).
4. Calculate the first hash using key_to_int function.
5. Calculate the second hash using secondhash function.
6. Start probing with an initial index equal to the first hash.
7. Loop until the slot matching the key is found:
    a. Increment the index by the second hash and take the modulo of TABLE_SIZE.
8. Mark the KeyValue structure at the found index as deleted.
```

9. Update the counters for the number of keys, occupied indices, and operations.
10. Return the index where the deletion happened.

-‘**get_load_factor’ Function:**
1. Calculate the load factor as the number of keys divided by TABLE_SIZE.
2. Update the load factor field in the HashTable structure.
3. Return the calculated load factor.

-‘**get_avg_probes’ Function:**
1. Calculate the average probes as the number of operations divided by the number of occupied indices.
2. Return the calculated average probes.

-‘**display’ Function**
1. Print a header for the hash table display.
2. Loop through each index in the array:
    a. If the slot is empty, print "NULL."
    b. If the slot is marked as deleted, print "deleted."
    c. If the slot contains a KeyValue structure, print its key and value.
3. Print a footer for the hash table display.

**PROGRAM:**

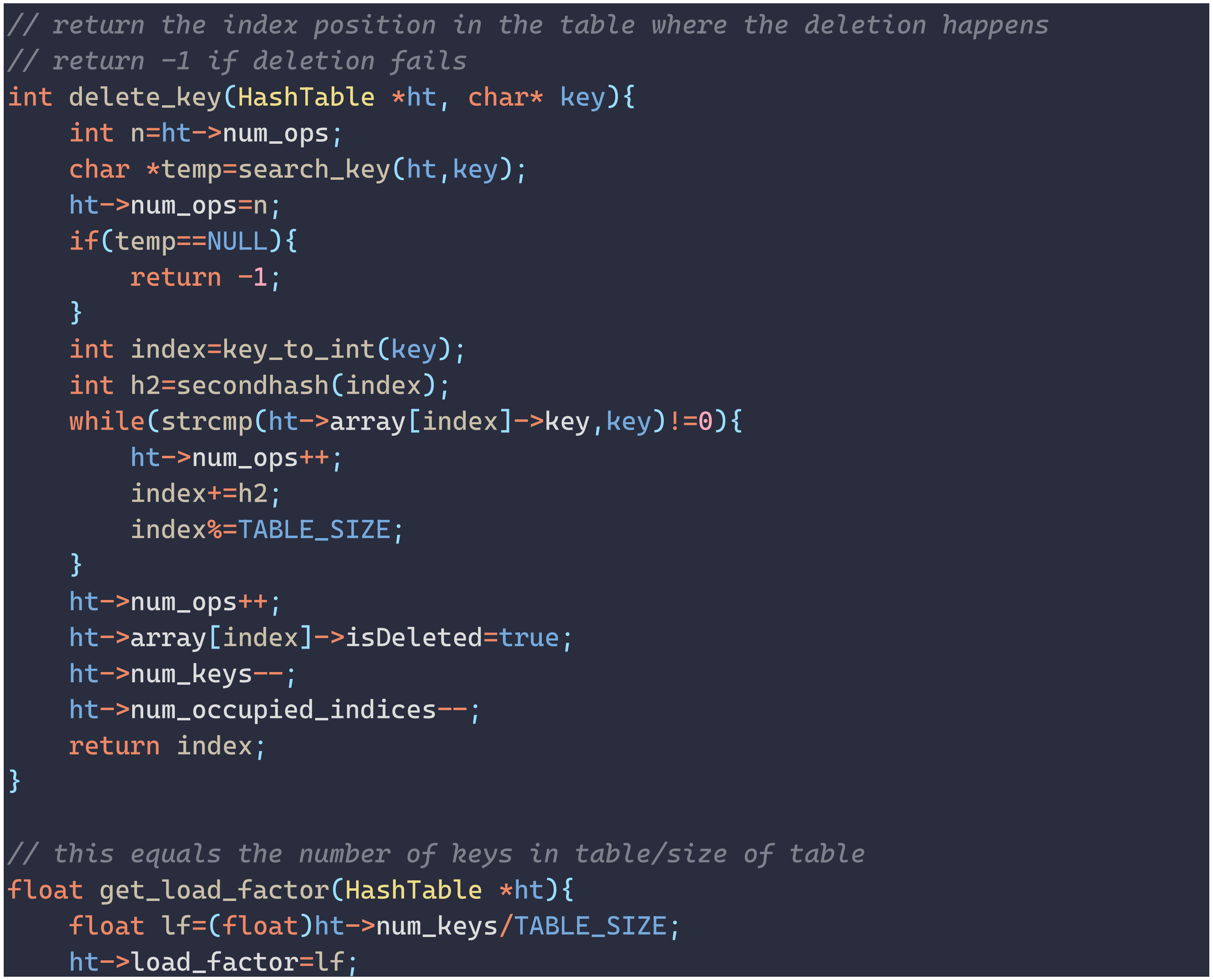
```c
/*
 * File: hashing_doublehashing.c
 * Author: Siddhartha Chandra
 * Email: siddhartha_chandra@spit.ac.in
 * Created: November 1, 2023
 * Description: This program implements hashing using using double hashing for collision resolution
 */
```

```c
// IMPORTANT: Use hash2(x) = 11 - (x % 11) as the 2nd hash function

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

#define TABLE_SIZE 23

typedef struct KeyValue {
    char *key;
    char *value;
    bool isDeleted;
} KeyValue;

typedef struct {
    KeyValue **array;
    int 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;
```

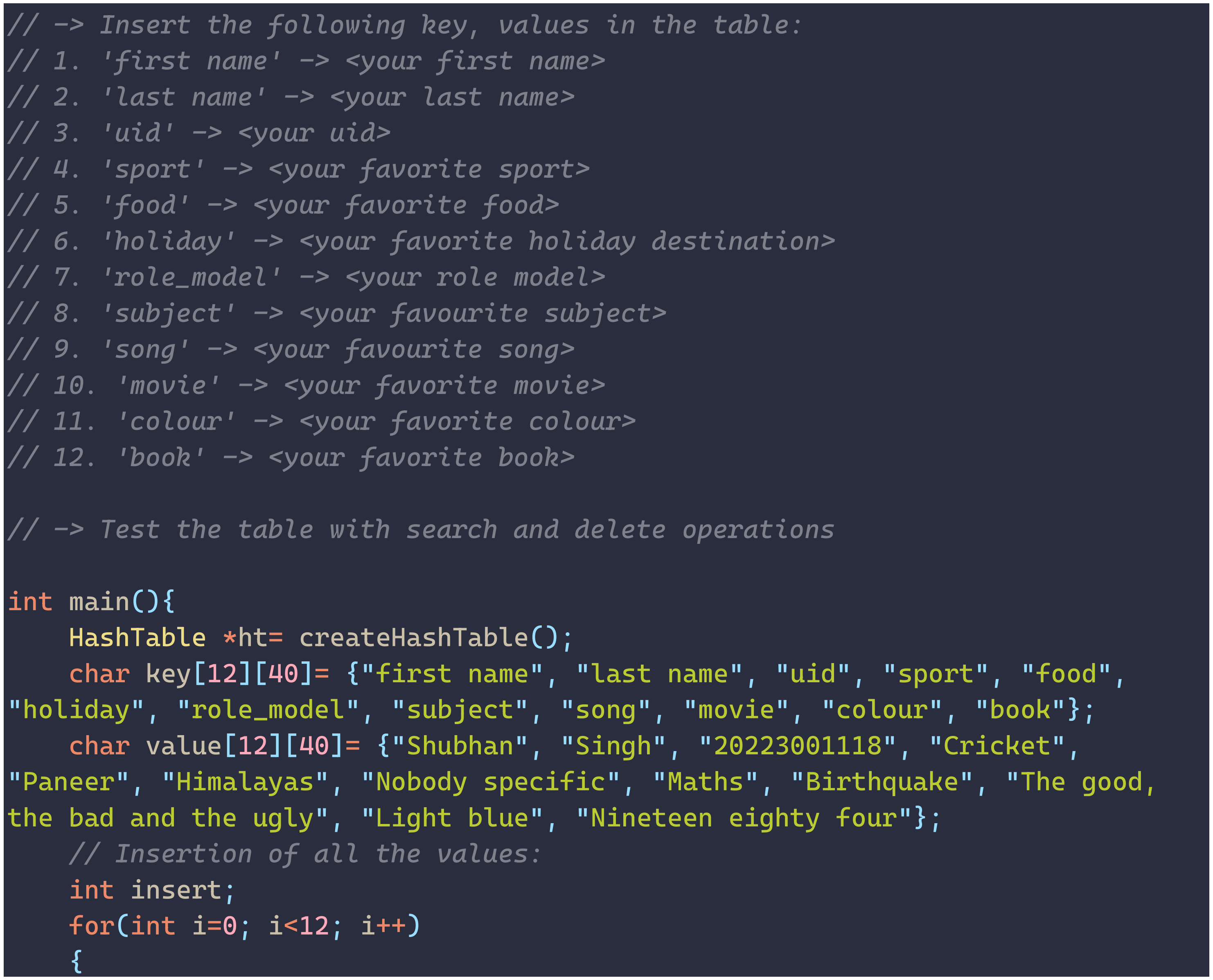
```c
} HashTable;


KeyValue *createKeyValue(char *key, char *value) {
    KeyValue* newKeyValue = malloc(sizeof(KeyValue));
    if (newKeyValue != NULL) {
        newKeyValue->key=malloc((strlen(key)+1)*sizeof(char));
        newKeyValue->value=malloc((strlen(value)+1)*sizeof(char));
        strcpy(newKeyValue->key,key);
        strcpy(newKeyValue->value,value);
        newKeyValue->isDeleted=false;
    }
    return newKeyValue;
}

HashTable* createHashTable() {
    HashTable* newTable = (HashTable*)malloc(sizeof(HashTable));
    newTable->array = (KeyValue **)malloc(TABLE_SIZE * sizeof(KeyValue *));
    for (int i=0; i<TABLE_SIZE; i++){
        newTable->array[i] = NULL;
    }
    newTable->size = TABLE_SIZE;
    newTable->load_factor = 0;
    newTable->num_keys = 0;
    newTable->num_occupied_indices = 0;
    newTable->num_ops = 0;
    return newTable;
```

```c
}


// use sum of ascii values to convert string to int
int key_to_int(char* key){
    int hash=0,ind=0;
    while(key[ind]!='\0'){
        hash+=((int)key[ind]+128);
        ind++;
    }
    hash%=TABLE_SIZE;
    return hash;
}

int secondhash(int x){
    return (11-(x%11));
}

// 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){
    if(ht->num_occupied_indices==TABLE_SIZE){
        return -1;
    }
    int h1=key_to_int(key);
    int retval;
    KeyValue* to_insert=createKeyValue(key,value);
    if(ht->array[h1]==NULL){
        ht->array[h1]=to_insert;
        ht->num_ops++;
        retval=h1;
    }
    else if(ht->array[h1]->isDeleted==true){
        ht->array[h1]->isDeleted=false;
        strcpy(ht->array[h1]->key,key);
        strcpy(ht->array[h1]->value,value);
        free(to_insert);
        ht->num_ops++;
        retval=h1;
    }
    else{
        int h2=secondhash(h1);
        int index=h1;
        while(ht->array[index]!=NULL){
            if(ht->array[index]->isDeleted==true){
                ht->array[index]->isDeleted=false;
                strcpy(ht->array[index]->key,key);
                strcpy(ht->array[index]->value,value);
                free(to_insert);
                retval=index;
                ht->num_ops++;
                goto wasdeleted;
            }
            index+=h2;
            index%=TABLE_SIZE;
            ht->num_ops++;
            if(index==h1){//if index becomes equal to initial value again, it
means it has gone into a loop
                return -1;
            }
        }
        ht->array[index]=to_insert;
        retval=index;
        ht->num_ops++;
    }
    wasdeleted:
    ht->num_keys++;
    ht->num_occupied_indices++;

    return retval;
}

// return the value of the key in the table
// return NULL if key not found
char *search_key(HashTable *ht, char* key){
    int h1=key_to_int(key);
    int index=h1;
    if(ht->array[index]==NULL){
        return NULL;
    }
    else if(strcmp(ht->array[index]->key,key)==0 && ht->array[index]-
```

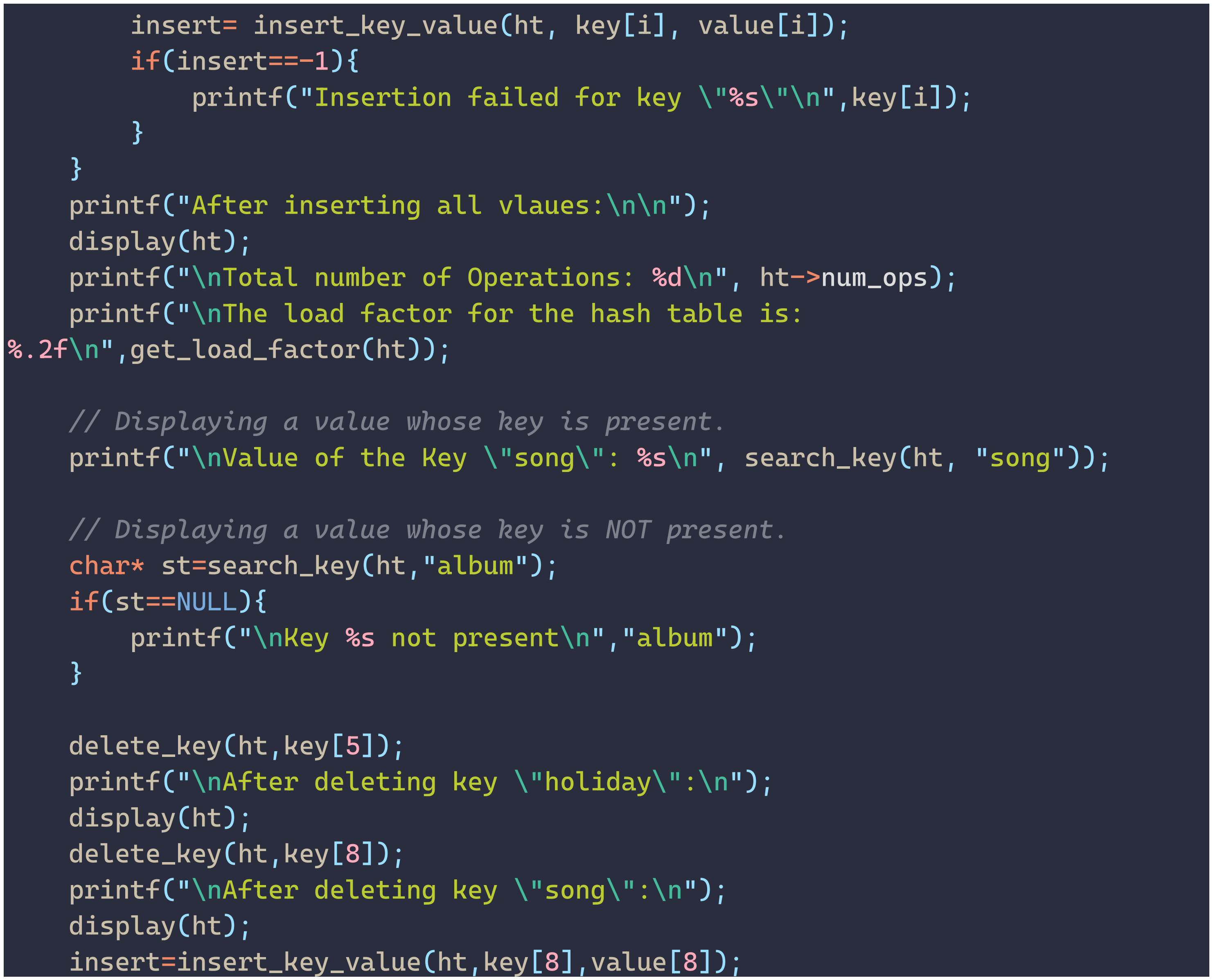
```c
>isDeleted==false){
        ht->num_ops++;
        return ht->array[index]->value;
    }
    else{
        int h2=secondhash(index);
        for(int i=0;i<TABLE_SIZE;i++){//if the item is not found after
table_size number of iterations, it means it doesn't exist as the function may
have gotten stuck in a loop
            index+=h2;
            index%=TABLE_SIZE;
            ht->num_ops++;
            if(ht->array[index]==NULL){
                return NULL;
            }
            else if(strcmp(ht->array[index]->key,key)==0 && ht->array[index]-
>isDeleted==false){
                return ht->array[index]->value;
            }
            else if(index==h1){
                return NULL;
            }
        }
    }
    return NULL;
}
```



```c
// return the index position in the table where the deletion happens
// return -1 if deletion fails
int delete_key(HashTable *ht, char* key){
    int n=ht->num_ops;
    char *temp=search_key(ht,key);
    ht->num_ops=n;
    if(temp==NULL){
        return -1;
    }
    int index=key_to_int(key);
    int h2=secondhash(index);
    while(strcmp(ht->array[index]->key,key)!=0){
        ht->num_ops++;
        index+=h2;
        index%=TABLE_SIZE;
    }
    ht->num_ops++;
    ht->array[index]->isDeleted=true;
    ht->num_keys--;
    ht->num_occupied_indices--;
    return index;
}

// this equals the number of keys in table/size of table
float get_load_factor(HashTable *ht){
    float lf=(float)ht->num_keys/TABLE_SIZE;
    ht->load_factor=lf;
    return lf;
}

// this equals the number of operations done so far/num of elems in table
float get_avg_probes(HashTable *ht){
    return (float)ht->num_ops/ht->num_occupied_indices;
}

// display hash table visually
void display(HashTable *ht){
    printf("displaying hash table:\n");
    printf("\nINDEX\t%-35s\t%-35s\n\n","KEY","VALUE");
    for(int i=0;i<TABLE_SIZE;i++){
        if(ht->array[i]==NULL){
            printf("%-5d\t%-35s\t%-35s\n",i,"NULL","NULL");
        }
        else if(ht->array[i]->isDeleted==true){
            printf("%-5d\t%-35s\t%-35s\n",i,"deleted","deleted");
        }
        else{
            printf("%-5d\t%-35s\t%-35s\n",i,ht->array[i]->key,ht->array[i]-
>value);
        }
    }
    printf("\n");
}
```



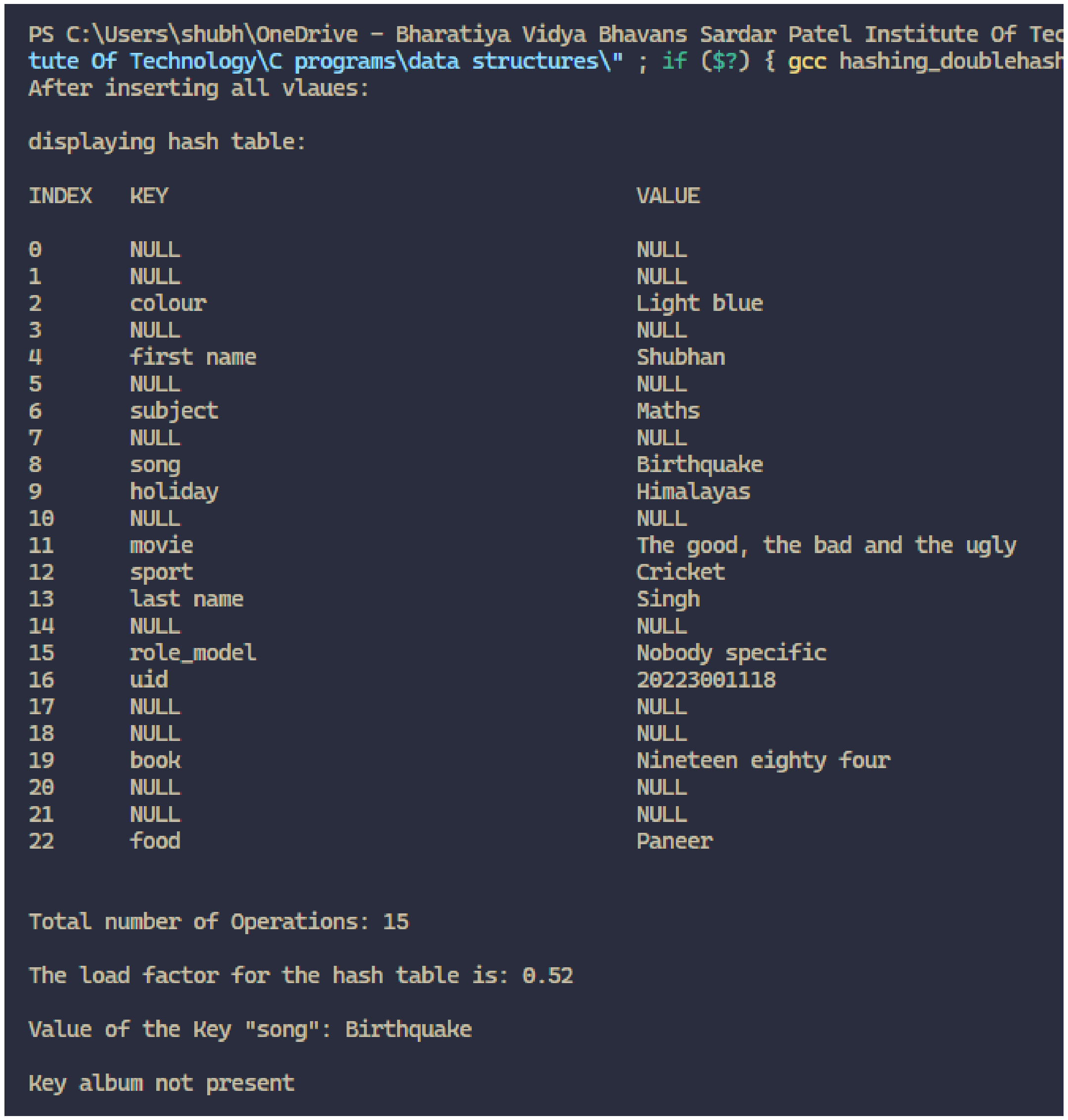
```c
// -> Insert the following key, values in the table:
// 1. 'first name' -> <your first name>
// 2. 'last name' -> <your last name>
// 3. 'uid' -> <your uid>
// 4. 'sport' -> <your favorite sport>
// 5. 'food' -> <your favorite food>
// 6. 'holiday' -> <your favorite holiday destination>
// 7. 'role_model' -> <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>

// -> Test the table with search and delete operations

int main(){
    HashTable *ht= createHashTable();
    char key[12][40]= {"first name", "last name", "uid", "sport", "food",
"holiday", "role_model", "subject", "song", "movie", "colour", "book"};
    char value[12][40]= {"Shubhan", "Singh", "20223001118", "Cricket",
"Paneer", "Himalayas", "Nobody specific", "Maths", "Birthquake", "The good,
the bad and the ugly", "Light blue", "Nineteen eighty four"};
    // Insertion of all the values:
    int insert;
    for(int i=0; i<12; i++)
    {
```



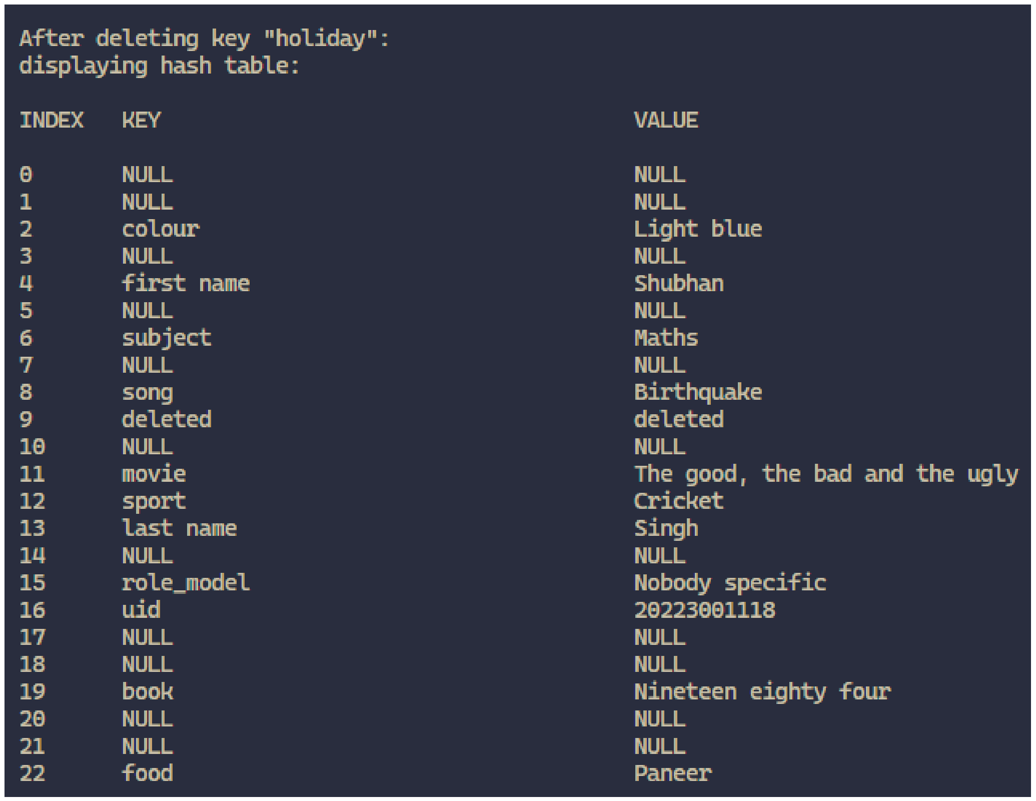
```
        insert= insert_key_value(ht, key[i], value[i]);
        if(insert==-1){
            printf("Insertion failed for key \"%s\"\n",key[i]);
        }
    }
    printf("After inserting all vlaues:\n\n");
    display(ht);
    printf("\nTotal number of Operations: %d\n", ht->num_ops);
    printf("\nThe load factor for the hash table is:
%.2f\n",get_load_factor(ht));

    // Displaying a value whose key is present.
    printf("\nValue of the Key \"song\": %s\n", search_key(ht, "song"));

    // Displaying a value whose key is NOT present.
    char* st=search_key(ht,"album");
    if(st==NULL){
        printf("\nKey %s not present\n","album");
    }

    delete_key(ht,key[5]);
    printf("\nAfter deleting key \"holiday\":\n");
    display(ht);
    delete_key(ht,key[8]);
    printf("\nAfter deleting key \"song\":\n");
    display(ht);
    insert=insert_key_value(ht,key[8],value[8]);
```

```c
    if(insert==-1){
        printf("Insertion failed for key \"%s\"\n",key[8]);
    }
    insert=insert_key_value(ht,key[5],value[5]);
    if(insert==-1){
        printf("Insertion failed for key \"%s\"\n",key[5]);
    }
    printf("After inserting deleted keys back:\n");
    display(ht);

return 0;
}
```

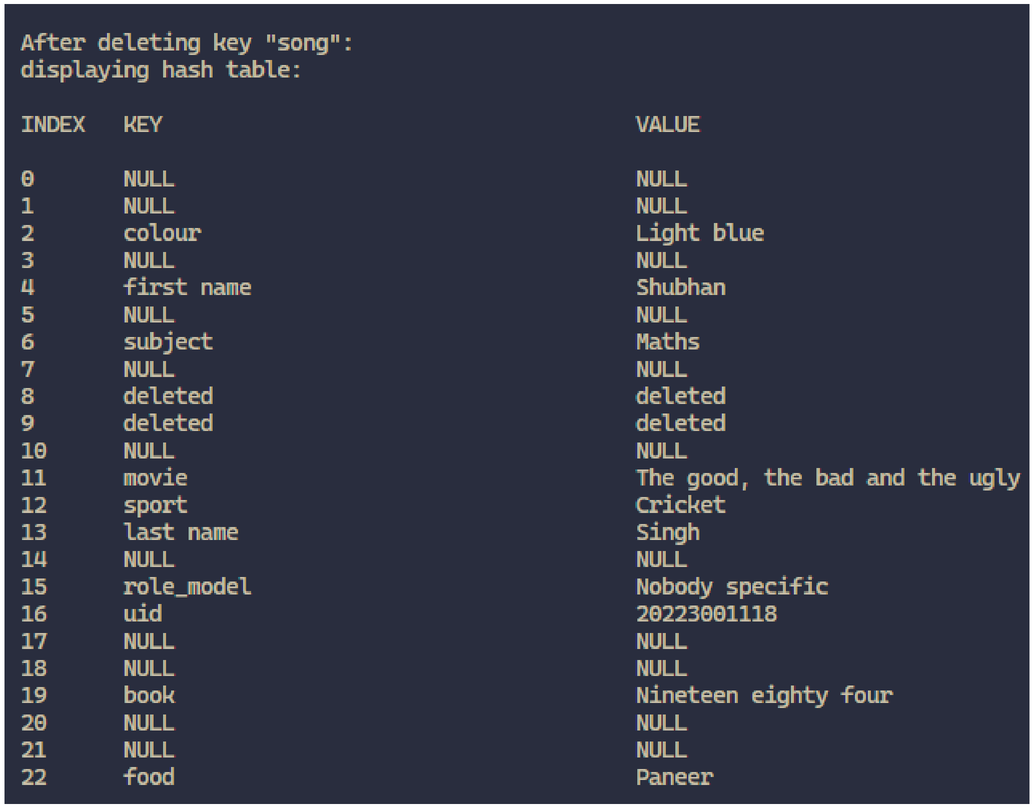
**RESULT:**



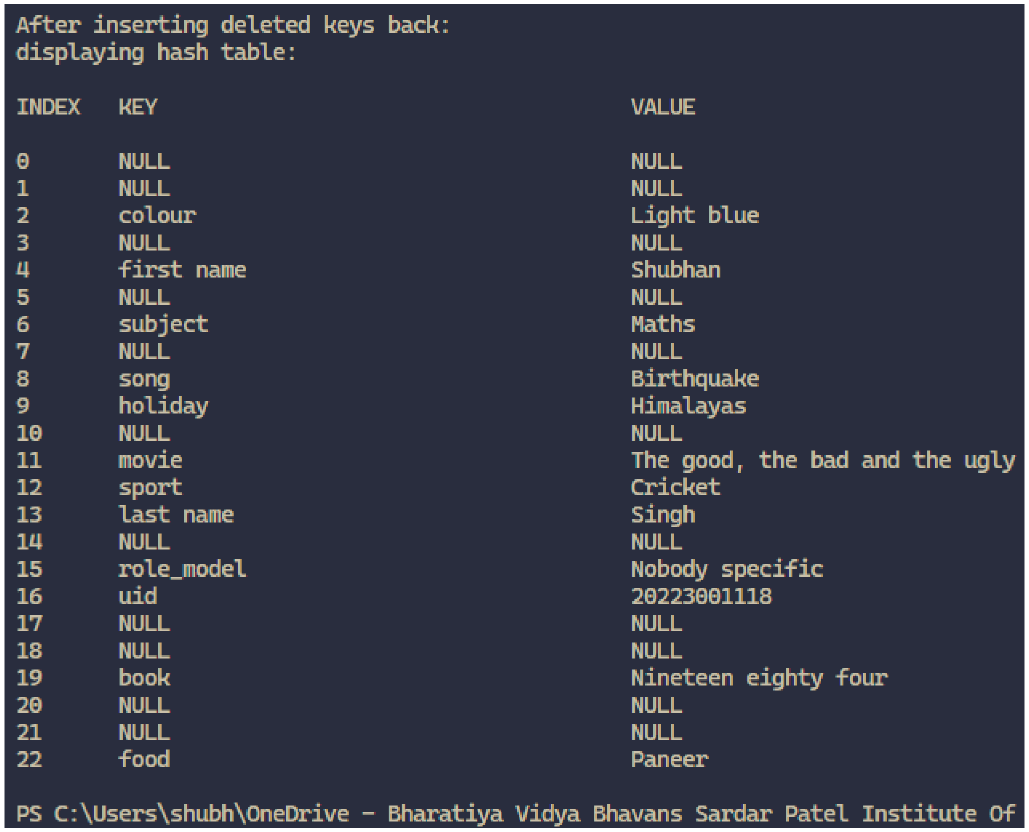
```
PS C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans Sardar Patel Institute Of Tec
tute Of Technology\C programs\data structures\" ; if ($?) { gcc hashing_doublehash
After inserting all vlaues:

displaying hash table:

INDEX   KEY                                       VALUE

0       NULL                                      NULL
1       NULL                                      NULL
2       colour                                    Light blue
3       NULL                                      NULL
4       first name                                Shubhan
5       NULL                                      NULL
6       subject                                   Maths
7       NULL                                      NULL
8       song                                      Birthquake
9       holiday                                   Himalayas
10      NULL                                      NULL
11      movie                                     The good, the bad and the ugly
12      sport                                     Cricket
13      last name                                 Singh
14      NULL                                      NULL
15      role_model                                Nobody specific
16      uid                                       20223001118
17      NULL                                      NULL
18      NULL                                      NULL
19      book                                      Nineteen eighty four
20      NULL                                      NULL
21      NULL                                      NULL
22      food                                      Paneer


Total number of Operations: 15

The load factor for the hash table is: 0.52

Value of the Key "song": Birthquake

Key album not present
```



```
After deleting key "holiday":
displaying hash table:

INDEX   KEY                                     VALUE

0       NULL                                    NULL
1       NULL                                    NULL
2       colour                                  Light blue
3       NULL                                    NULL
4       first name                              Shubhan
5       NULL                                    NULL
6       subject                                 Maths
7       NULL                                    NULL
8       song                                    Birthquake
9       deleted                                 deleted
10      NULL                                    NULL
11      movie                                   The good, the bad and the ugly
12      sport                                   Cricket
13      last name                               Singh
14      NULL                                    NULL
15      role_model                              Nobody specific
16      uid                                     20223001118
17      NULL                                    NULL
18      NULL                                    NULL
19      book                                    Nineteen eighty four
20      NULL                                    NULL
21      NULL                                    NULL
22      food                                    Paneer
```



```
After deleting key "song":
displaying hash table:

INDEX   KEY                                     VALUE

0       NULL                                    NULL
1       NULL                                    NULL
2       colour                                  Light blue
3       NULL                                    NULL
4       first name                              Shubhan
5       NULL                                    NULL
6       subject                                 Maths
7       NULL                                    NULL
8       deleted                                 deleted
9       deleted                                 deleted
10      NULL                                    NULL
11      movie                                   The good, the bad and the ugly
12      sport                                   Cricket
13      last name                               Singh
14      NULL                                    NULL
15      role_model                              Nobody specific
16      uid                                     20223001118
17      NULL                                    NULL
18      NULL                                    NULL
19      book                                    Nineteen eighty four
20      NULL                                    NULL
21      NULL                                    NULL
22      food                                    Paneer
```



```
After inserting deleted keys back:
displaying hash table:

INDEX   KEY                                     VALUE

0       NULL                                    NULL
1       NULL                                    NULL
2       colour                                  Light blue
3       NULL                                    NULL
4       first name                              Shubhan
5       NULL                                    NULL
6       subject                                 Maths
7       NULL                                    NULL
8       song                                    Birthquake
9       holiday                                 Himalayas
10      NULL                                    NULL
11      movie                                   The good, the bad and the ugly
12      sport                                   Cricket
13      last name                               Singh
14      NULL                                    NULL
15      role_model                              Nobody specific
16      uid                                     20223001118
17      NULL                                    NULL
18      NULL                                    NULL
19      book                                    Nineteen eighty four
20      NULL                                    NULL
21      NULL                                    NULL
22      food                                    Paneer

PS C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans Sardar Patel Institute Of
```

| | |
|---|---|
| **THEORY:** | Hashing is a technique used in computer science to map data of arbitrary size to fixed-size values, typically integers. The goal is to create a hash function that efficiently distributes data across a fixed-size array, allowing for quick retrieval and storage of information. Hashing is commonly used in various data structures, such as hash tables, to provide constant-time average complexity for insertion, deletion, and retrieval operations.<br>In a hash table, data is stored in an array, and a hash function is applied to each key to determine its index in the array. However, collisions can occur when two different keys produce the same hash value, leading to a conflict in the array index. Addressing collisions is crucial to maintaining the efficiency of hash table operations.<br>**Collision Prevention Strategies:** |

1. **Separate Chaining:**
    - In separate chaining, each array index contains a linked list or another data structure to handle multiple values that hash to the same index.
    - When a collision occurs, the new key-value pair is added to the linked list at the corresponding index.
    - This strategy ensures that all values with the same hash value are stored together, minimizing search time within the linked list.
2. **Open Addressing:**
    - Open addressing involves finding an alternative location within the hash table when a collision occurs.
    - Different probing techniques, such as linear probing, quadratic probing, or double hashing, are used to search for the next available slot.
    - Linear probing checks the next slot, quadratic probing checks slots based on a quadratic function, and double hashing uses a second hash function to determine the step size.

**Double Hashing:**

Double hashing is a specific open addressing strategy that addresses collisions by applying a second hash function to calculate the step size between probes. The idea is to resolve collisions by systematically searching for the next available slot.

**Double Hashing Collision Resolution Algorithm:**

1. **Primary Hashing:**
    - Calculate the primary hash using the initial hash function.
    - Determine the initial index for the key.
2. **Secondary Hashing:**
    - Calculate the secondary hash using a second hash function.
    - Use the secondary hash value as the step size for probing.
3. **Probing:**
    - Start probing from the initial index.
    - If the slot is occupied, move to the next index based on the secondary hash function.

| | |
|---|---|
| | • Repeat until an empty slot is found.<br>4. **Insertion or Retrieval:**<br>• For insertion, place the key-value pair in the first available empty slot.<br>• For retrieval, continue probing until finding the key or an empty slot.<br>**Advantages of Double Hashing:**<br>• Provides a deterministic way to find the next available slot, reducing clustering.<br>• Can efficiently handle a wide range of load factors.<br>• Avoids the primary clustering issues associated with linear probing.<br>**Considerations:**<br>• The second hash function should produce different values for different inputs.<br>• Care must be taken to handle scenarios where the probing sequence completes a full loop. |
| **CONCLUSION:** | Double hashing is a collision resolution technique employed in hash tables to address clustering issues. It involves using two hash functions—the first one determines the initial index, and the second one dictates the step size for probing. This approach reduces primary clustering, providing a systematic and deterministic way to find the next available slot during collision resolution. |