Experiment No. 9

Name : Manjiri Chavande	**DSE SY COMPS**
Division: B batch-A	**UID: 2023301003**

• Problem Statement

Implement Hashing using Linear Probing

• Theory

Hashing is a technique that maps data to a fixed-size array, called a hash table, using a hash function. It enables quick data retrieval based on keys, reducing search time to nearly constant. Collision resolution methods handle situations where different keys produce the same hash value, such as using separate chaining (linked lists) or **open addressing** (probing techniques). **Separate chaining** links elements with the same hash to a linked list, while open addressing finds alternative locations for colliding elements. Common open addressing methods include **linear probing**, **quadratic probing**, **and double hashing**. These methods ensure efficient storage and retrieval of data, minimizing the impact of collisions on the overall performance of the hash table.

Linear probing is a collision resolution method that handles this by sequentially searching for the next available index in the array. Step-by-step explanation of linear probing:

1. Compute the hash value of the key using a hash function.
2. If the computed index is empty, store the key-value pair there.
3. If the index is occupied, search for the next available index by incrementing the index in a linear manner.
4. Continue this process until an empty index is found or the entire table is traversed.
5. Insert the key-value pair at the first empty index found.
6. During search, follow the same linear probing process until the key is found or an empty index is encountered.
7. For deletion, mark the key-value pair as deleted without removing it from the array.
8. Ensure the probing wraps around to the beginning of the array if the end is reached.
9. Monitor the load factor to avoid table overflow.
10. Handle resizing of the table if the load factor exceeds a certain threshold.

- **How to calculate the hash key?**

Let's take hash table size as 7.
size = 7

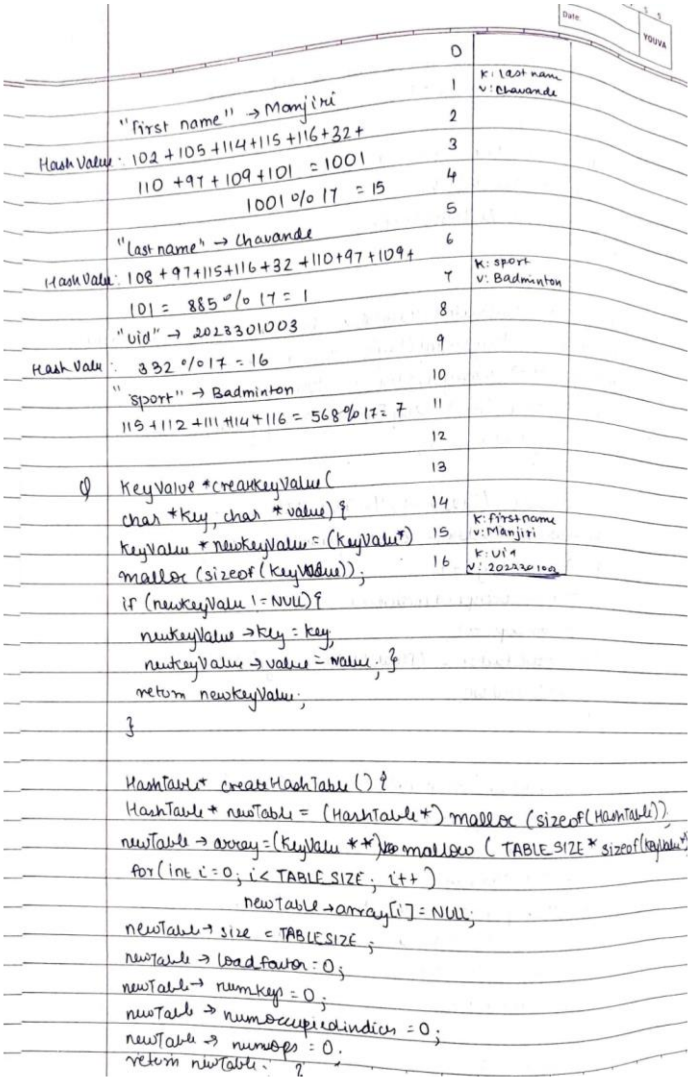
arr[size];
Formula to calculate key is,
key = element % size
If we take modulo of number with N, the remainder will always be 0 to N - 1.
Exactly array index also starts from 0 and ends with index N -1. So we can easily store elements in array index.

• Algorithm

1. **CreateKey values function:**
   a. **Dynamically allocate memory**
   b. **Assign newValue->key = key; and newValueKey->value =value**
   c. **Return the created newKeyValue pair**
2. **Create hashTable function:**
   a. **Dynamically allocate memory for newHashTable and the array inside it**
   b. **Interate through array and set all the values as NULL ;**
   c. **Set all the properties to 0;**
   d. **Return the created hash table.**
3. **convert a string key to an integer index**
   a. **iterate through the key until '\0' end of string is met**
   b. **Sum up the ASCII values of characters in the key**
   c. **Return the Modulo operation to ensure the index is within the table size**
4. **Insert a value**
   a. **Goto step 3 get the index**
   b. **Iterate thorught the array till it is not null**
   c. **Compare the current key and the key previously present at that particular index ,if it is true that means key alraey exists return -1;**
   d. **find the next available slot**
   e. **hkey = key% TABLE_SIZE Increment all the other properties , calculate the load factor**
   f. **Return the index.**
5. **Seraching**
   a. **Hashtable is an array of size = TABLE_SIZEStep**
   b. **1: Read the value to be searched, keyStep**
   c. **2: let i = 0**
   d. **(key+1)% TABLE_SIZE get next index**
   e. **compute the index at which the key can be found index = (hkey+ i) % TABLE_SIZE**
   f. **if the element at that index is same as the search value then print element found and STOP**
   g. **else   step 4: i = i+1**
   h. **if i < TABLE_SIZE then go to step 4**
6. **Delete**
   a. **Goto step 3 get the index , count = 0**

b. **While the array is not null and count <17**
c. **Check of the key exists if true free the index and decrement the num_keys and increment the operation by 1;**
d. **Return the index**
e. **Index = (key+1)% TABLE_SIZE**
f. **Count++**

• Solution



Date: YOUVA

"first name" → Manjiri

Hash Value : 102 + 105 + 114 + 115 + 116 + 32 +
110 + 97 + 109 + 101 = 1001
1001 % 17 = 15

"last name" → Chavande

Hash Value : 108 + 97 + 115 + 116 + 32 + 110 + 97 + 109 +
101 = 885 % 17 = 1

"uid" → 2023301003

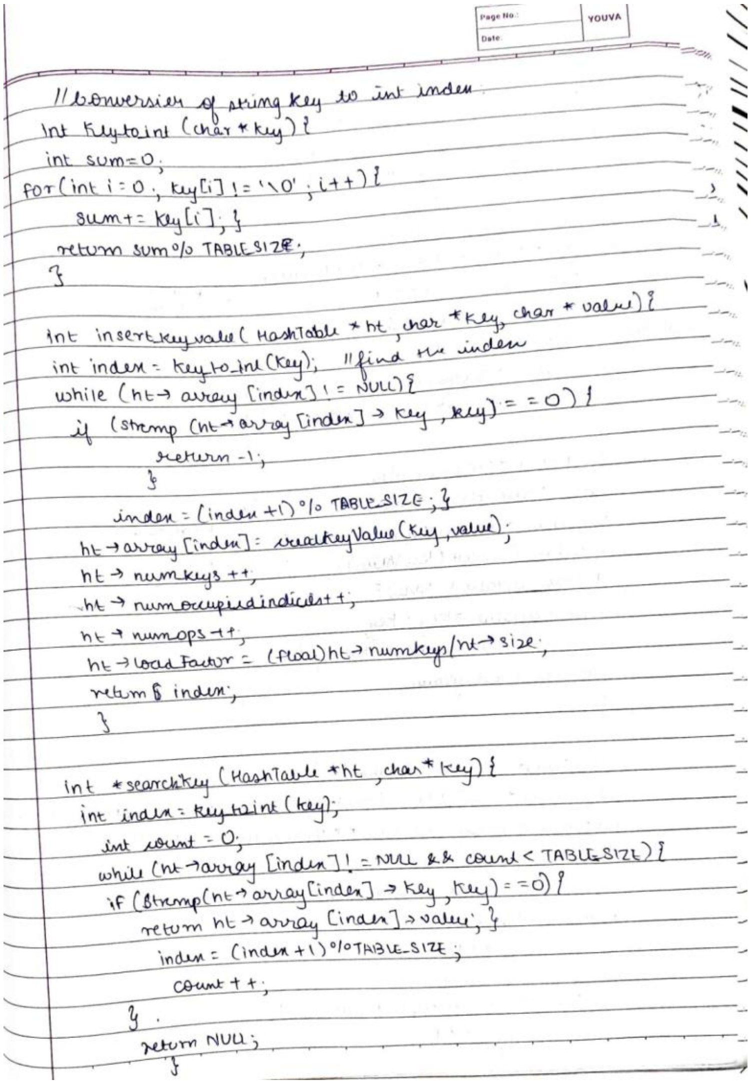
Hash Value : 332 % 17 = 16

"sport" → Badminton

115 + 112 + 111 + 114 + 116 = 568 % 17 = 7

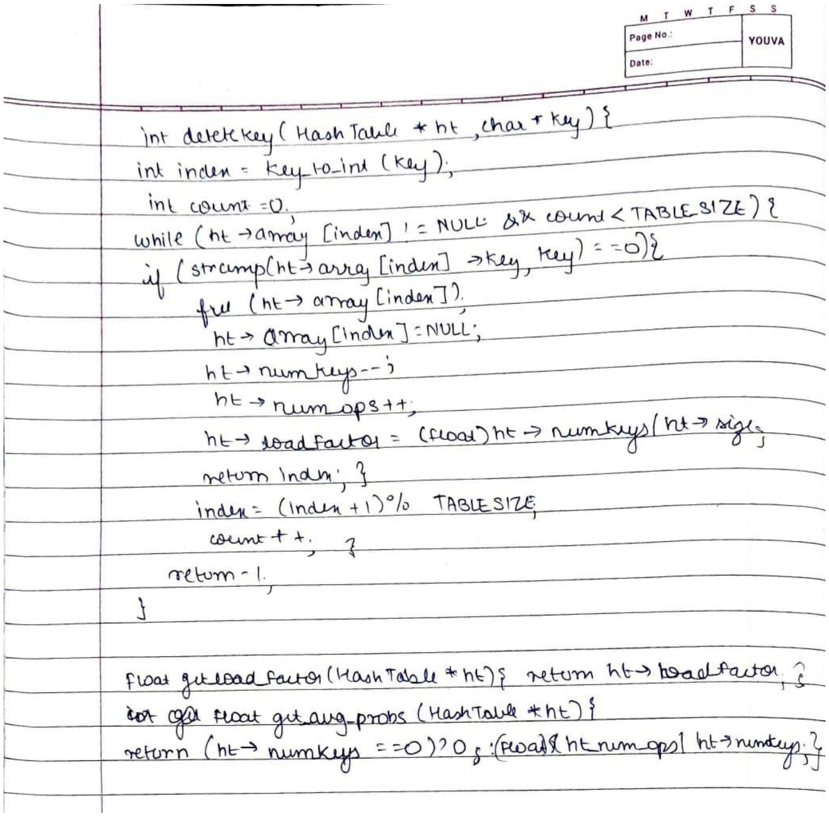
0	
1	K: last name V: Chavande
2	
3	
4	
5	
6	
7	K: sport V: Badminton
8	
9	
10	
11	
12	
13	
14	
15	K: firstname V: Manjiri
16	K: Uid V: 202330100

```
KeyValue *creatKeyValue(
char *key, char *value){
KeyValue * newKeyValue = (KeyValue*)
malloc (sizeof(KeyValue));
if (newKeyValue != NULL){
  newKeyValue → key = key;
  newKeyValue → value = value; }
 return newKeyValue;
}

HashTable* createHashTable(){
HashTable * newTable = (HashTable*) malloc (sizeof(HashTable));
newTable → array = (KeyValue **) mallow ( TABLE SIZE * sizeof(KeyValue*)
 for(int i=0; i< TABLE SIZE; i++)
                newTable → array[i] = NULL;
newTable → size = TABLESIZE;
newTable → load factor = 0;
newTable → numkeys = 0;
newTable → numoccupiedindices = 0;
newTable → numops = 0.
return newTable;  }
```



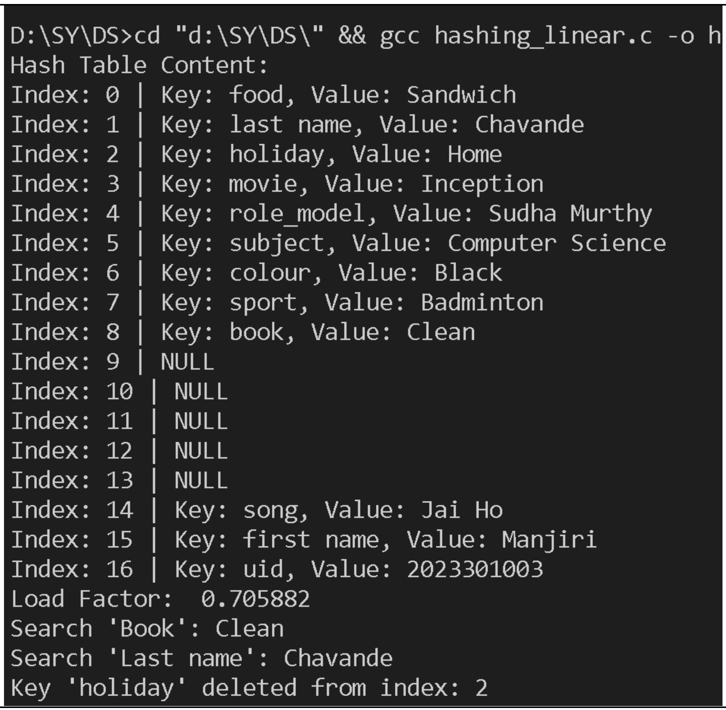
Page No.: Date: YOUVA

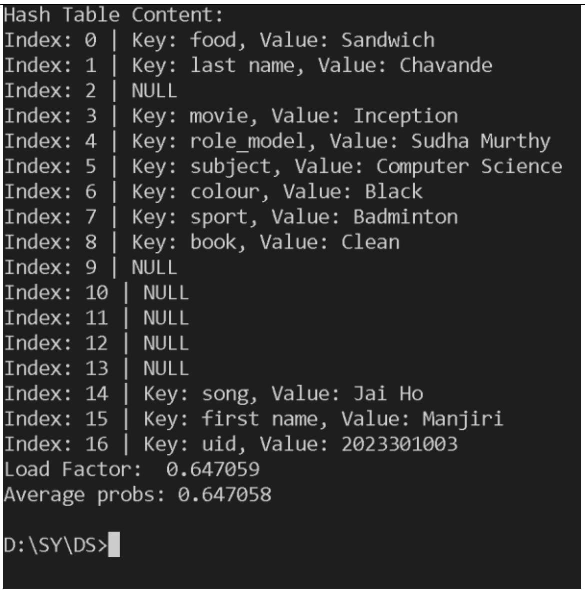
```
//conversion of string key to int index
int keytoint (char * key){
 int sum=0;
for(int i=0; key[i] != '\0'; i++){
    sum+= key[i]; }
 return sum% TABLESIZE;
 }

int insertkeyvalue( HashTable * ht, char *key, char * value){
 int index = key_to_int(key);  //find the index
 while (ht→ array [index] != NULL){
   if (strcmp (ht→array [index] → key , key) = =0) {
          return -1;
         }
      index = (index +1) % TABLE_SIZE ; }
  ht→array [index]= createkeyValue (key, value);
  ht → numkeys ++;
  ht → numoccupiedindices++;
  ht → numops ++;
  ht → loadFactor = (float)ht→ numkeys/ht→ size;
  return index;
   }

int *searchkey (HashTable *ht , char* key) {
 int index = keytoint (key);
   int count = 0;
  while (ht→array [index] != NULL && count < TABLESIZE) {
   if (strcmp(ht→array[index] → key , key) = =0) {
      return ht→array [index]→value; }
        index = (index +1) %TABLE_SIZE ;
          count ++;
     } .
     return NULL;
      }
```



```
int deletekey( HashTable * ht ,char * key) {
int index = key_to_int (key);
 int count =0;
while (ht→array [index] != NULL && count < TABLESIZE) {
if (strcmp(ht→array [index] →key, key) = =0){
    free (ht→ array [index]);
     ht→ array[index] = NULL;
     ht→ numkeys--;
      ht → num_ops++;
     ht→ loadfactor = (float) ht → numkeys/ ht→ size;
     return index; }
    index= (index +1)% TABLESIZE;
     count ++;   }
  return -1;
}

float getloadfactor(HashTable * ht){ return ht→ loadfactor; }
~~int get~~ float get_avg_probs (HashTable * ht){
return (ht→ numkeys ==0)? 0 : (float) ht_num_ops / ht→ numkeys; }
```

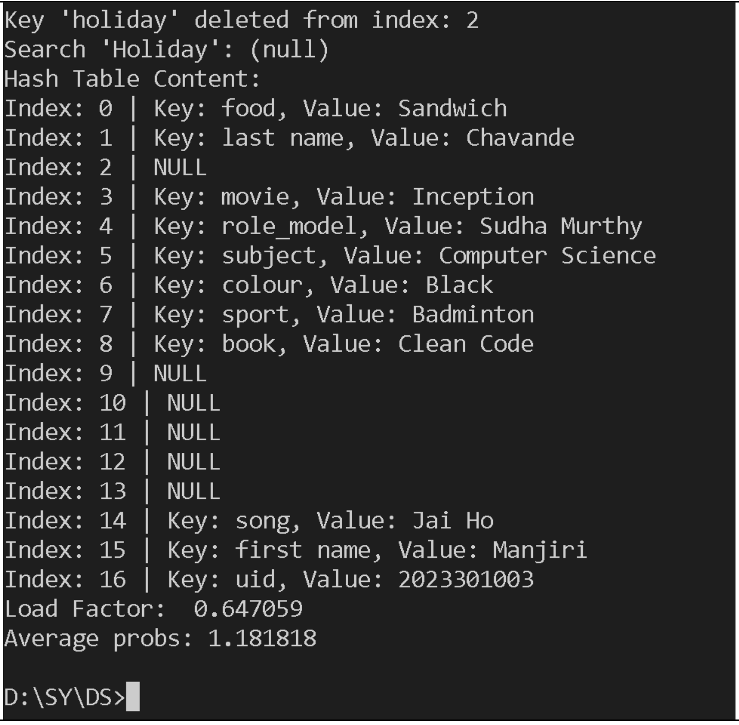
Output

<table>
<tr>
<td><b>When all the values are inserted , and deletion of holiday key:
Size of the array is 17</b></td>
<td>

<pre>
D:\SY\DS>cd "d:\SY\DS\" &amp;&amp; gcc hashing_linear.c -o h
Hash Table Content:
Index: 0 | Key: food, Value: Sandwich
Index: 1 | Key: last name, Value: Chavande
Index: 2 | Key: holiday, Value: Home
Index: 3 | Key: movie, Value: Inception
Index: 4 | Key: role_model, Value: Sudha Murthy
Index: 5 | Key: subject, Value: Computer Science
Index: 6 | Key: colour, Value: Black
Index: 7 | Key: sport, Value: Badminton
Index: 8 | Key: book, Value: Clean
Index: 9 | NULL
Index: 10 | NULL
Index: 11 | NULL
Index: 12 | NULL
Index: 13 | NULL
Index: 14 | Key: song, Value: Jai Ho
Index: 15 | Key: first name, Value: Manjiri
Index: 16 | Key: uid, Value: 2023301003
Load Factor: 0.705882
Search 'Book': Clean
Search 'Last name': Chavande
Key 'holiday' deleted from index: 2
</pre>
</td>
</tr>
</table>



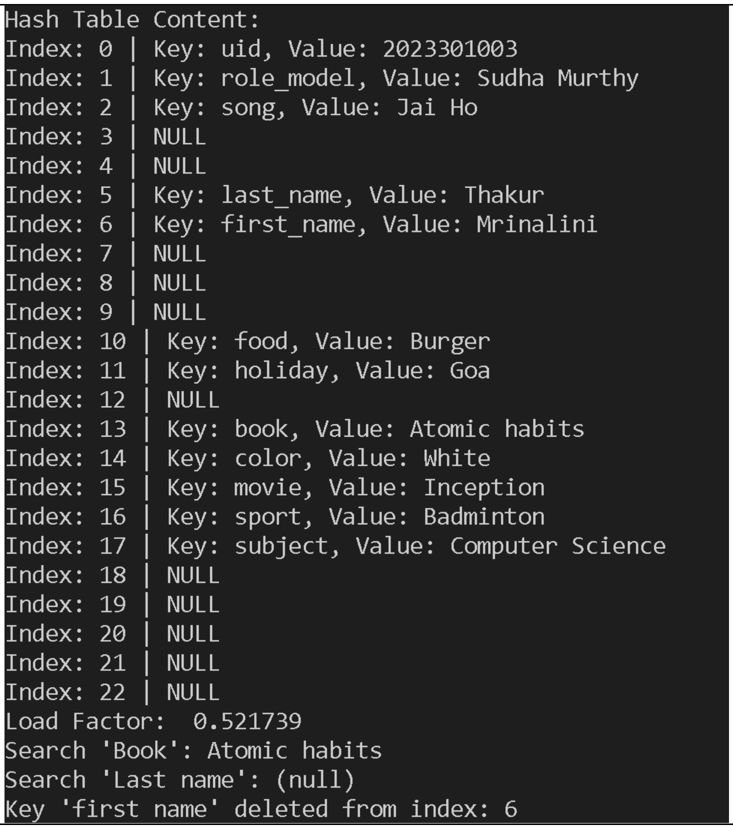
```
Hash Table Content:
Index: 0 | Key: food, Value: Sandwich
Index: 1 | Key: last name, Value: Chavande
Index: 2 | NULL
Index: 3 | Key: movie, Value: Inception
Index: 4 | Key: role_model, Value: Sudha Murthy
Index: 5 | Key: subject, Value: Computer Science
Index: 6 | Key: colour, Value: Black
Index: 7 | Key: sport, Value: Badminton
Index: 8 | Key: book, Value: Clean
Index: 9 | NULL
Index: 10 | NULL
Index: 11 | NULL
Index: 12 | NULL
Index: 13 | NULL
Index: 14 | Key: song, Value: Jai Ho
Index: 15 | Key: first name, Value: Manjiri
Index: 16 | Key: uid, Value: 2023301003
Load Factor:  0.647059
Average probs: 0.647058

D:\SY\DS>
```

When the value searching does not exists



```
Key 'holiday' deleted from index: 2
Search 'Holiday': (null)
Hash Table Content:
Index: 0 | Key: food, Value: Sandwich
Index: 1 | Key: last name, Value: Chavande
Index: 2 | NULL
Index: 3 | Key: movie, Value: Inception
Index: 4 | Key: role_model, Value: Sudha Murthy
Index: 5 | Key: subject, Value: Computer Science
Index: 6 | Key: colour, Value: Black
Index: 7 | Key: sport, Value: Badminton
Index: 8 | Key: book, Value: Clean Code
Index: 9 | NULL
Index: 10 | NULL
Index: 11 | NULL
Index: 12 | NULL
Index: 13 | NULL
Index: 14 | Key: song, Value: Jai Ho
Index: 15 | Key: first name, Value: Manjiri
Index: 16 | Key: uid, Value: 2023301003
Load Factor:  0.647059
Average probs: 1.181818

D:\SY\DS>
```

• Test Case

Testing for different values Array size :23	

```
Hash Table Content:
Index: 0 | Key: uid, Value: 2023301003
Index: 1 | Key: role_model, Value: Sudha Murthy
Index: 2 | Key: song, Value: Jai Ho
Index: 3 | NULL
Index: 4 | NULL
Index: 5 | Key: last_name, Value: Thakur
Index: 6 | Key: first_name, Value: Mrinalini
Index: 7 | NULL
Index: 8 | NULL
Index: 9 | NULL
Index: 10 | Key: food, Value: Burger
Index: 11 | Key: holiday, Value: Goa
Index: 12 | NULL
Index: 13 | Key: book, Value: Atomic habits
Index: 14 | Key: color, Value: White
Index: 15 | Key: movie, Value: Inception
Index: 16 | Key: sport, Value: Badminton
Index: 17 | Key: subject, Value: Computer Science
Index: 18 | NULL
Index: 19 | NULL
Index: 20 | NULL
Index: 21 | NULL
Index: 22 | NULL
Load Factor:  0.521739
Search 'Book': Atomic habits
Search 'Last name': (null)
Key 'first name' deleted from index: 6
```

• Conclusion

Thus we have successfully implemented linear probing.