Bharatiya Vidya Bhavan's
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Autonomous Institute Affiliated to University of Mumbai)
Munshi Nagar, Andheri (W), Mumbai – 400 058.
COMPS Department

Experiment	9
Aim	**Hashing Techniques: Write a program to implement Hash Table for the given input and solve the collision using using quadratic probing and linear probing and double hashing.**
Name	David Daniels
UID	2023300038
Class	Div -A
Batch	C
Date of Submission	23-10-24

Theory	**Hashing** is a technique used in data structures that efficiently stores and retrieves data in a way that allows for quick access. It involves mapping data to a specific index in a hash table using a **hash function** that enables fast retrieval of information based on its key. This method is commonly used in databases, **c**aching systems, and various programming applications to optimize search and retrieval operations. The great thing about hashing is, we can achieve all three operations (search, insert and delete) in O(1) time on average. There are many applications of hashing,such as: 1) Database indexing: Hashing is used to index and retrieve data efficiently in databases and other data storage systems. 2) Dictionaries : To implement a dictionary so that we can quickly search a word 3) Password storage: Hashing is used to store passwords securely by applying a hash function to the password and storing the hashed result, rather than the plain text password. 4) Network Routing: Determining the best path for data packets 5) Cryptography: Hashing is used in cryptography to generate digital signatures, message authentication codes (MACs), and key derivation functions. 6) Load balancing: Hashing is used in load-balancing algorithms, such as consistent hashing, to distribute requests to servers in a network. 7) Blockchain: Hashing is used in blockchain technology, such as the proof-of-work algorithm, to secure the integrity and consensus of the blockchain. 8) Image processing: Hashing is used in image processing applications, such as perceptual hashing, to detect and prevent image duplicates and modifications.

9) File comparison: Hashing is used in file comparison algorithms, such as the MD5 and SHA-1 hash functions, to compare and verify the integrity of files.
10) Caching: Storing frequently accessed data for faster retrieval. For example browser caches, we can use URL as keys and find the local storage of the URL.
11) Symbol Tables: Mapping identifiers to their values in programming languages
12) Associative Arrays: Associative arrays are nothing but hash tables only. Commonly SQL library functions allow you retrieve data as associative arrays so that the retrieved data in RAM can be quickly searched for a key.

Collisions and How to Handle Them
Two or more keys can generate same hash values sometimes. This is called a collision. The new key should also be placed in the hash Somewhere. A collision can be handled using various techniques.

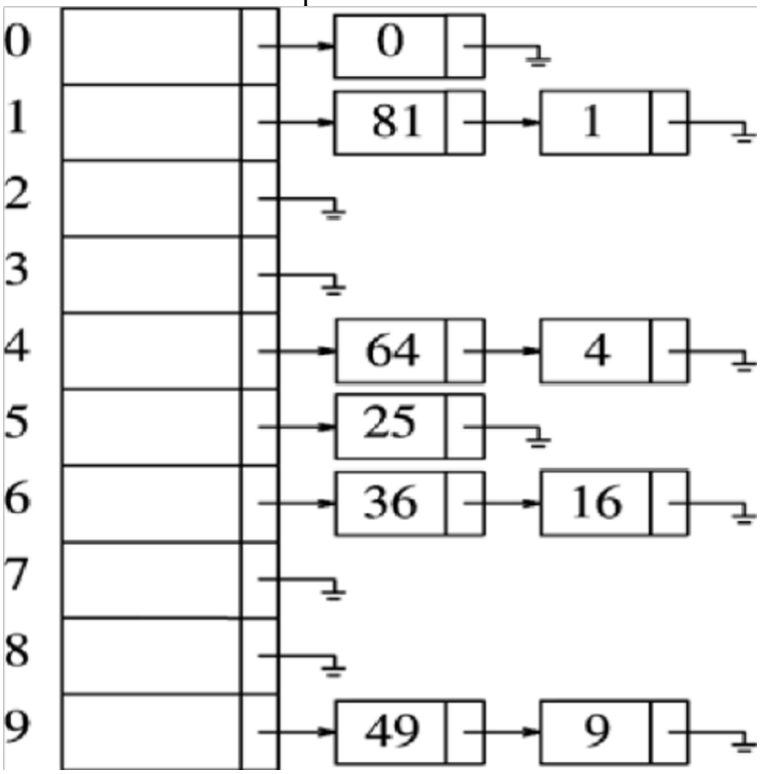
There are two major Types of Collision Resolution Techniques
1) **Separate Chaining Technique**
2) **Open Addressing technique**

Open Addressing technique
In this method, the values are all stored **in** the hash table itself. If collision occurs, we look for availability in the next spot generated by an algorithm. The table size at all times should be greater than the number of keys. *It is used when there is space restrictions, like in embedded processors.*
Point to note in delete operations, the deleted slot needs to be marked in some way so that during searching, we don't stop probing at empty slots.
Types of Open Addressing:

1. Linear Probing: We linearly probe/look for next slot. If slot [hash(x)%size] is full, we try [hash(x)%size+1]. If that is full too, we try [hash(x)%size+2]…until an available space is found. Linear Probing has the best cache performance but downside includes primary and secondary clustering.

2. Quadratic Probing: We look for i^2th iteration. If slot [hash(x)%size] is full, we try [(hash(x)+1*1)%size]. If that is also full, we try [(hash(x)+2*2)%size]…until an available space is found. Secondary clustering might arise here and there is no guarantee of finding a slot in this approach.

3. Double Hashing: We use a second hash function hash2(x) and look for i*hash2(x) slot. If slot [hash(x)%size] is full, we try [(hash(x)+1*hash2(x))%size]. If that is full too, we try [(hash(x)+2*hash2(x))%size]…until an available space is found.

	No primary or secondary clustering but lot more computation here. **Separate Chaining Technique** The idea is to make each cell of the hash table point to a linked list of records that have the same hash function values. It is simple but requires additional memory outside the table. In this technique, the worst case occurs when all the values are in the same index or linked list, making the search complexity linear (n=length of the linked list). *This method should be used when we do not know how many keys will be there or how frequently the insert/delete operations will take place.* To maintain the O(1) time of insertions, we make the new value as head of the linked list of the particular index. 
Algorithm	**1) Linear Probing** 1. **Calculate Initial Hash Index**: • Use the hash_fn function to compute the initial index rem for the given key based on the hash table's size. 2. **Check for Duplicate**: • If the value at h->arr[rem] is equal to key, print a message indicating a duplicate exists, and exit the function. 3. **Check for Empty Slot**: • If the status at h->status[rem] indicates it's empty ('E'): o Insert key into h->arr[rem]. o Update the status at h->status[rem] to 'F' (indicating it's filled).

4. **Handle Collision**:

- If the slot is not empty, print a message indicating a collision has occurred.
- Initialize a probe counter probe_no starting at 1.

5. **Linear Probing**:

- Enter a loop that continues until an empty slot is found:
  - Check again if h->arr[rem] is equal to key. If it is, print a duplicate message and exit.
  - Increment rem to check the next slot (using modulo to wrap around the table).
  - Print the current probe number and the new index being checked.
  - Increment the probe_no.

6. **Insert the Key**:

- Once an empty slot is found (when h->status[rem] is 'E'):
  - Insert key into h->arr[rem].
  - Update the status to 'F'.

2) Quadratic Probing

1. **Initialize Variables**:

- Create an ecount variable to keep track of the number of attempts made to insert the key.
- Compute the initial index rem using the hash_fn function.

2. **Check for Duplicate**:

- If the value at h->arr[rem] matches key, print a message indicating a duplicate exists, and exit the function.

3. **Print Initial Probe**:

- Print the details of the initial probing attempt using the calculated index rem.

4. **Check for Empty Slot**:

- If the status at h->status[rem] indicates it's empty ('E'):
  - Insert key into h->arr[rem].
  - Update the status at h->status[rem] to 'F'.

5. **Handle Collision with Quadratic Probing**:

- If the slot is occupied, print a message indicating a collision has occurred.
- Initialize a temporary variable temp starting at 1.

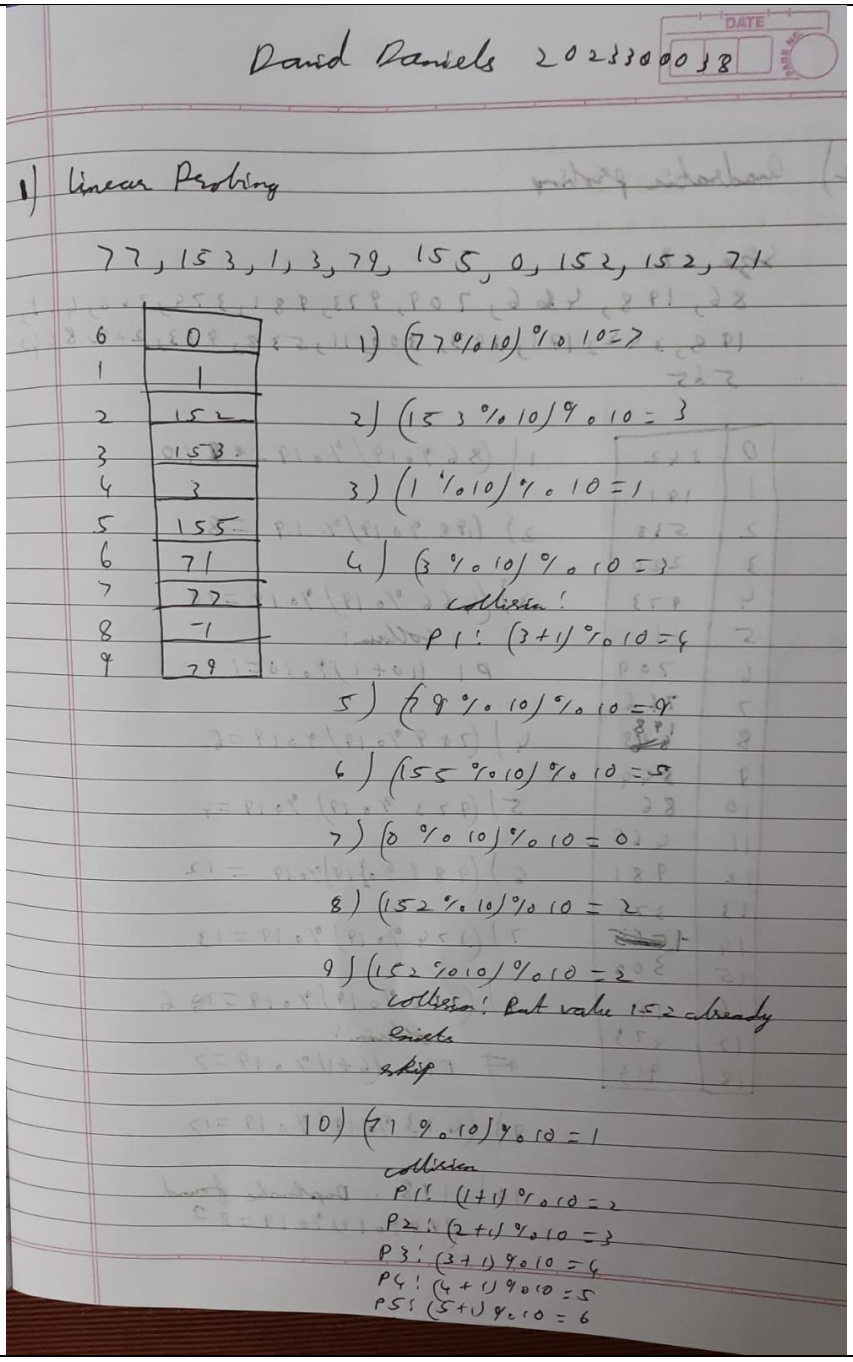
6. **Quadratic Probing Loop**:

- Enter a loop that continues until an empty slot is found or the number of attempts (ecount) exceeds twice the size of the hash table:
  - Check again if h->arr[rem] matches key. If it does, print a duplicate message and exit.
  - Calculate the new index using the formula: rem = (rem + (temp * temp)) % h->size.
  - Print the current probing attempt details.
  - Increment temp, and increase ecount by 1.

7. **Check for Infinite Loop**:

	• If ecount exceeds h->size * 2 - 2, print a warning message indicating potential infinite looping. 8. **Insert the Key**: • Once an empty slot is found, insert key into h->arr[rem] and update the status to 'F'. **3) Double Hashing** 1. **Calculate Initial Hash Index**: • Compute the initial index rem for the given key using the hash_fn function. • Store the original index in orem for later reference. • Calculate a secondary index rem2 using the hash_fn1 function to assist in probing. 2. **Initialize Attempt Counter**: • Set ecount to zero to keep track of the number of probing attempts. 3. **Check for Duplicate**: • If the value at h->arr[rem] is equal to key, print a message indicating a duplicate exists and exit the function. 4. **Print Initial Probe Information**: • Print details of the initial probing attempt using the calculated index rem. 5. **Check for Empty Slot**: • If the status at h->status[rem] indicates it's empty ('E'): o Insert key into h->arr[rem]. o Update the status at h->status[rem] to 'F' (indicating it's filled). 6. **Handle Collision with Double Hashing**: • If the slot is occupied, print a message indicating a collision has occurred. • Initialize a temporary variable temp starting at 1. 7. **Probing Loop**: • Enter a loop that continues until an empty slot is found or the number of attempts (ecount) exceeds twice the size of the hash table: o Calculate the new index using the formula: rem = (orem + temp * rem2) % h->size. o Check if h->arr[rem] matches key. If it does, print a duplicate message and exit. o Print the current probing attempt details, including the values used in the calculation. o Increment temp and increase ecount by 1. 8. **Check for Infinite Loop**: • If ecount exceeds h->size * 2 - 2, print a warning message indicating potential infinite looping. 9. **Insert the Key**: • Once an empty slot is found, insert key into h->arr[rem] and update the status to 'F'.

Problem Solving



David Daniels 202330038

1) Linear Probing

77, 153, 1, 3, 79, 155, 0, 152, 152, 71

0	0
1	1
2	152
3	153
4	3
5	155
6	71
7	77
8	-1
9	79

1) (77 % 10) % 10 = 7

2) (153 % 10) % 10 = 3

3) (1 % 10) % 10 = 1

4) (3 % 10) % 10 = 3
collision!
P1: (3+1) % 10 = 4

5) (79 % 10) % 10 = 9

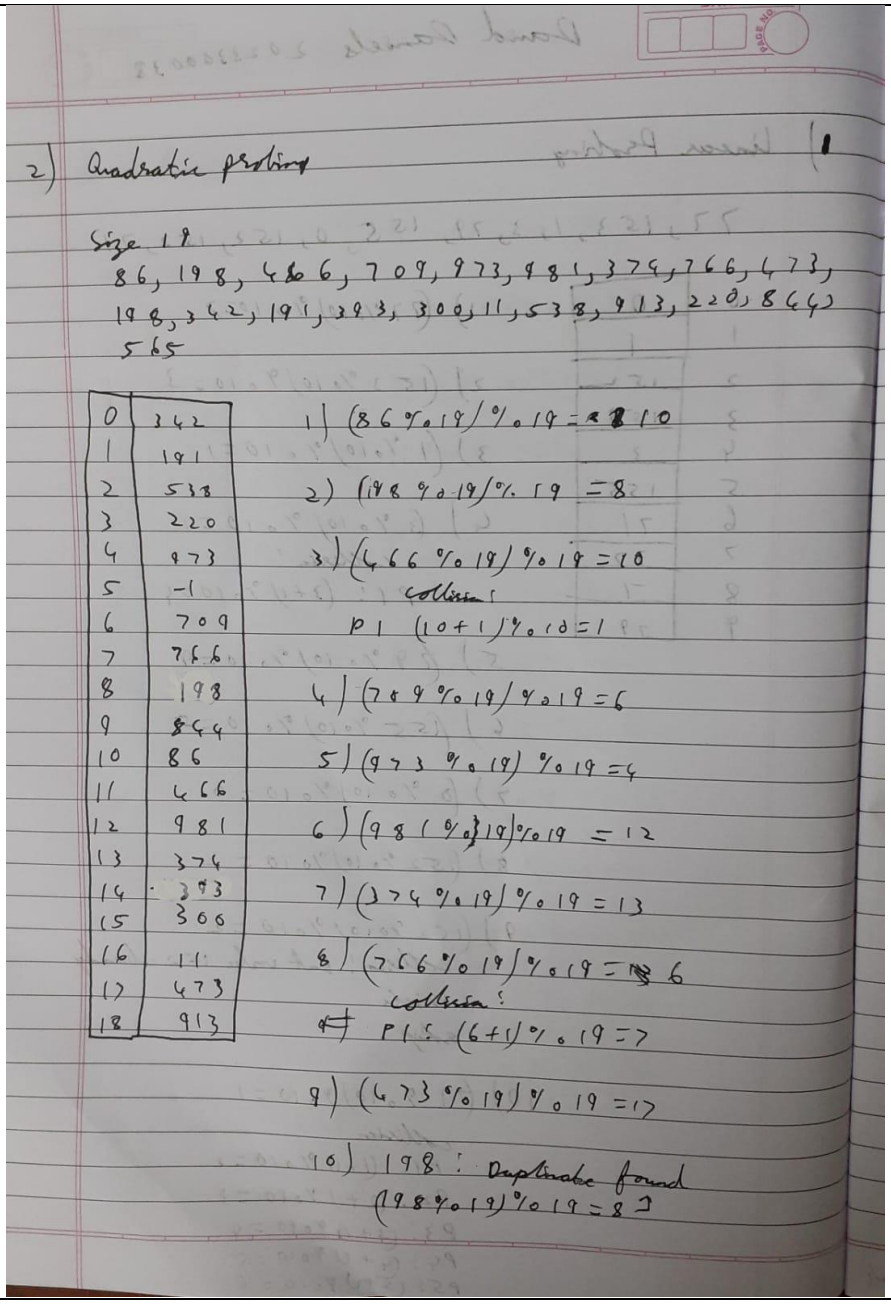
6) (155 % 10) % 10 = 5

7) (0 % 10) % 10 = 0

8) (152 % 10) % 10 = 2

9) (152 % 10) % 10 = 2
collision! But value 152 already exists
skip

10) (71 % 10) % 10 = 1
collision
P1: (1+1) % 10 = 2
P2: (2+1) % 10 = 3
P3: (3+1) % 10 = 4
P4: (4+1) % 10 = 5
P5: (5+1) % 10 = 6



2) Quadratic probing

Size 19

86, 198, 466, 709, 973, 981, 374, 766, 473, 198, 342, 191, 393, 300, 11, 538, 913, 220, 864, 565

0	342
1	191
2	538
3	220
4	973
5	-1
6	709
7	766
8	198
9	864
10	86
11	466
12	981
13	374
14	393
15	300
16	11
17	473
18	913

1) (86 % 19) % 19 = 10

2) (198 % 19) % 19 = 8

3) (466 % 19) % 19 = 10
collision!
P1 (10+1) % 19 = 1

4) (709 % 19) % 19 = 6

5) (973 % 19) % 19 = 4

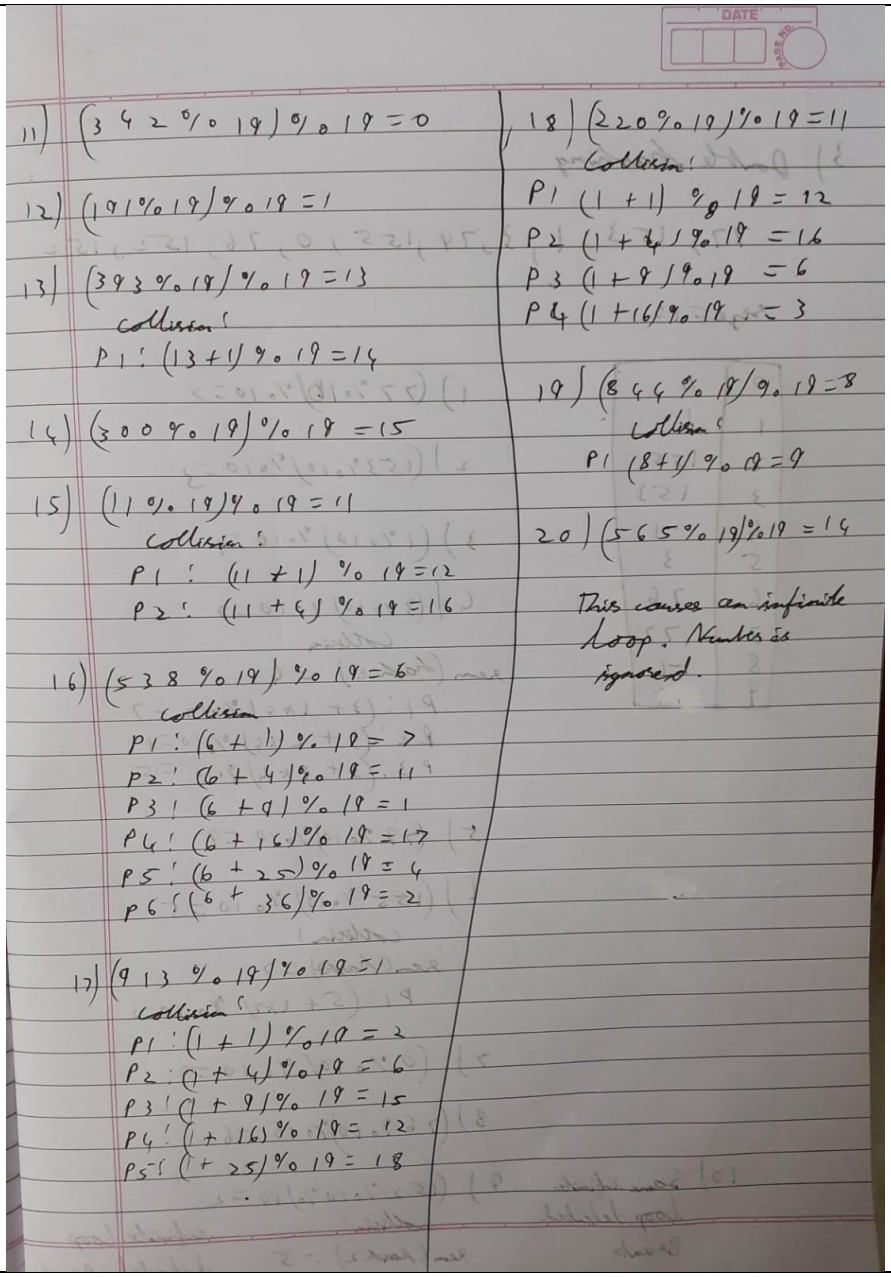
6) (981 % 19) % 19 = 12

7) (374 % 19) % 19 = 13

8) (766 % 19) % 19 = 6
collision!
P1: (6+1) % 19 = 7

9) (473 % 19) % 19 = 17

10) 198 : Duplicate found
(198 % 19) % 19 = 8



11) (342 % 19) % 19 = 0

12) (191 % 19) % 19 = 1

13) (393 % 19) % 19 = 13
collision:
P1: (13 + 1) % 19 = 14

14) (300 % 19) % 19 = 15

15) (11 % 19) % 19 = 11
collision:
P1: (11 + 1) % 19 = 12
P2: (11 + 4) % 19 = 16

16) (538 % 19) % 19 = 6
collision
P1: (6 + 1) % 19 = 7
P2: (6 + 4) % 19 = 11
P3: (6 + 9) % 19 = 1
P4: (6 + 16) % 19 = 17
P5: (6 + 25) % 19 = 4
P6: (6 + 36) % 19 = 2

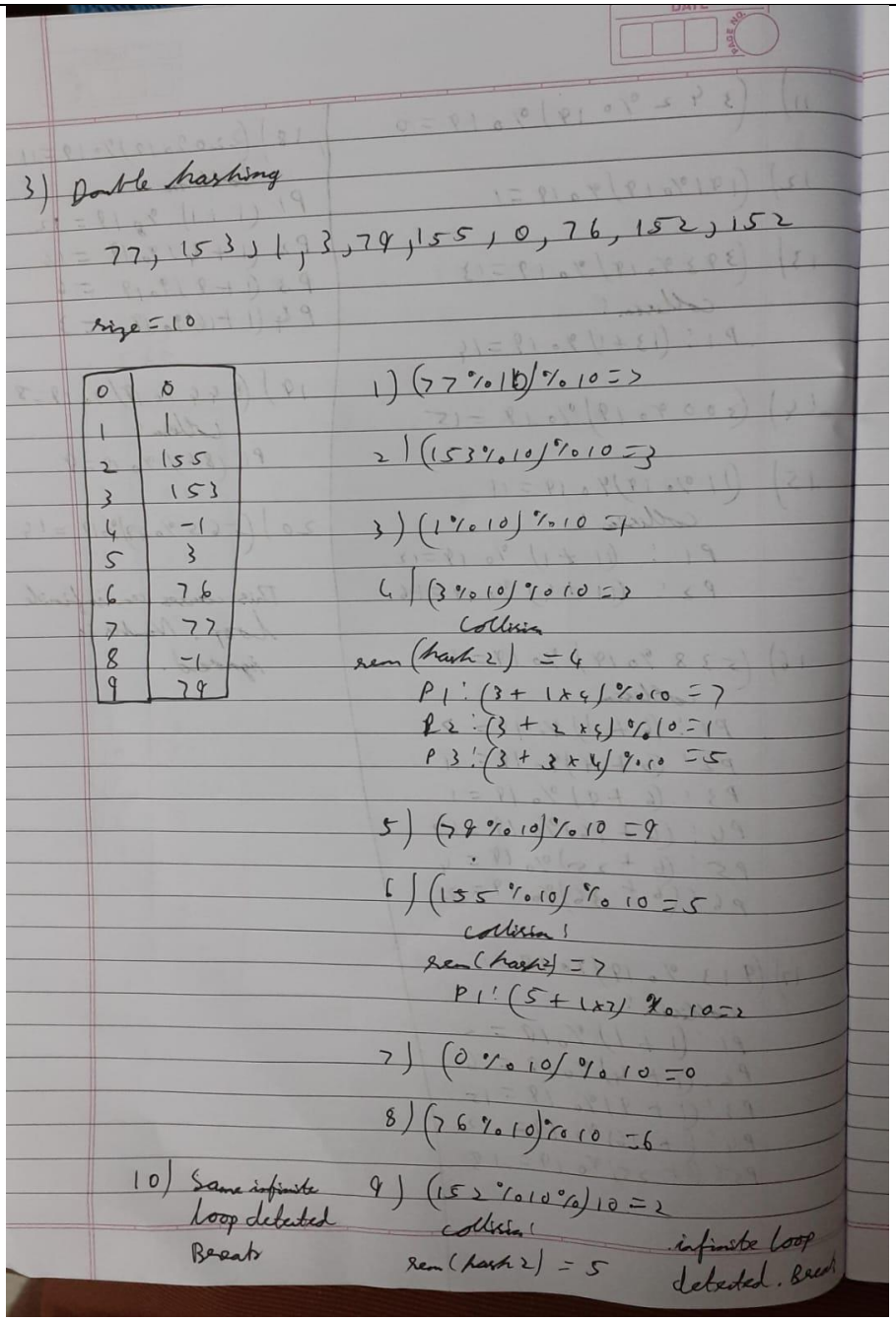
17) (913 % 19) % 19 = 1
collision:
P1: (1 + 1) % 19 = 2
P2: (1 + 4) % 19 = 6
P3: (1 + 9) % 19 = 15
P4: (1 + 16) % 19 = 12
P5: (1 + 25) % 19 = 18

18) (220 % 19) % 19 = 11
collision:
P1 (1 + 1) % 19 = 12
P2 (1 + 4) % 19 = 16
P3 (1 + 9) % 19 = 6
P4 (1 + 16) % 19 = 3

19) (844 % 19) % 19 = 8
collision:
P1 (8 + 1) % 19 = 9

20) (565 % 19) % 19 = 14

This causes an infinite loop. Numbers is ignored.



Program(Code)

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

typedef struct hash
{
   int * arr;
   int size;
   char * status;
}hash;

int hash_fn (hash * h ,int key)
{
```

```
    return key%h->size;
}

void insert_lin(hash * h, int key)
{
    int rem =hash_fn(h,key);
    printf("\n Applying Probe 0: (%d %% %d) %% %d = %d \n", key , h-
>size, h->size, rem);
    if (h->arr[rem]==key)
    {
        printf("You Have a Duplicate: No need to insert: \n");
        return;
    }
    if (h->status[rem]=='E')
    {
        h->arr[rem]=key;
        h->status[rem]='F';
    }
    else
    {
        printf("A Collision has occured: \n");
        int probe_no=1;
        //apply probing
        while (h->status[rem]!='E')
        {
            if (h->arr[rem]==key)
            {
                printf("You Have a Duplicate: No need to insert: \n");
                return;
            }

            rem= (rem+probe_no)%h->size;
            printf("Applying Probe %d: (%d+%d) %% %d = %d
\n",probe_no, key,probe_no, h->size,  rem);
            probe_no++;
        }
        h->arr[rem]=key;
        h->status[rem]='F';
    }
    printf("Element %d has been inserted at %d \n",key,rem);
}

void insert_quad(hash * h, int key)
{
    int ecount=0;
    int rem =hash_fn(h,key);
    if (h->arr[rem]==key)
    {
        printf("You Have a Duplicate: No need to insert: \n");
        return;
```

```
    }
    int orem=rem;
    printf("\n Applying Probe 0: (%d %% %d) %% %d = %d \n", key , h->size, h->size, rem);
    if (h->status[rem]=='E')
    {
       h->arr[rem]=key;
       h->status[rem]='F';
    }
    else
    {
       printf("A Collision has occured: \n");
       int temp=1;
       while (h->status[rem]!='E'&& ecount<h->size*2)
       {
          if (h->arr[rem]==key)
          {
             printf("You Have a Duplicate: No need to insert: \n");
             return;
          }
       rem=(rem + (temp*temp))%h->size;
       printf("Applying Probe %d: ( %d + %d ) %% %d = %d \n",temp,orem, temp*temp , h->size,  rem);
       temp++;
       ecount++;
       }
       if (ecount>(h->size*2)-2)
       {
       printf("Element %d is going in infinite loop: ",key);
       return;
       }
       h->arr[rem]=key;
       h->status[rem]='F';
    }

    printf("Element %d has been inserted at %d \n",key,rem);
}

int hash_fn1 (hash *h , int key)
{
    return 7- ((key%5));
}

void insert_doub(hash * h, int key)
{
    int rem =hash_fn(h,key);
    int orem=rem;
    int rem2 = hash_fn1(h,key);
    int ecount=0;
    if (h->arr[rem]==key)
```

```
    {
        printf("You Have a Duplicate: No need to insert: \n");
        return;
    }
    printf("\n Applying Probe 0: (%d %% %d) %% %d = %d \n", key , h->size, h->size, rem);

    if (h->status[rem]=='E')
    {
        h->arr[rem]=key;
        h->status[rem]='F';
    }
    else
    {
        //apply probing
        printf("A Collision has occured: \n");
        int temp=1;
        while (h->status[rem]!='E' && ecount<h->size*2)
        {

            rem= (orem + temp*rem2) %h->size;
            if (h->arr[rem]==key)
            {
                printf("You Have a Duplicate: No need to insert: \n");
                return;
            }
            printf("Applying Probe %d: ( %d + %d*%d ) %% %d = %d \n",temp ,orem,temp,rem2,h->size,rem);
            temp++;
            ecount++;
        }

        if (ecount>(h->size*2)-2)
        {
        printf("Element %d is going in infinite loop: \n",key);
        return;
        }
        h->arr[rem]=key;
        h->status[rem]='F';
    }



    printf("Element %d has been inserted at %d \n",key,rem);
}

void print_hash(hash* h)
{
    printf("\n Index  Element Stored  Status \n");
    for (int i = 0; i < h->size; i++)
```

```
    {
        printf("%d  \t    %d\t \t %c \n",i , h->arr[i], h->status[i]);
    }
}

void choice(hash * h)
{
    printf("Enter 1 to Do linear probing \nEnter 2 to do Quadratic
probing: \nEnter 3 to do Double Hashing \n");
    int num=0;
    scanf("%d",&num);
    switch (num)
    {
    case 1:
    {
        printf("Welcome to linear probing \n Enter 1 to insert \n Enter 2 to
print the entire Table \n Enter 0 to Exit \n");
        while (true)
        {
            int choice=0;
            printf("Enter choice \n");
            scanf("%d",&choice);
            if (choice==0)
            {
                printf("\nGoodbye\n");
                break;
            }

            switch (choice)
            {
            case 1:
            {
                printf("Enter number to insert: ");
                scanf("%d", &num);
                insert_lin(h,num);
                break;
            }
            case 2:
            {
                print_hash(h);
                break;
            }

            default:
            {
                printf("Error");
                break;
            }
            }
        }
```

```
      break;
   }

   case 2:
   {
      printf("Welcome to Quadratic probing \n Enter 1 to insert \n Enter
2 to print the entire Table \n Enter 0 to Exit \n");
      while (true)
      {
         int choice=0;
         printf("Enter choice \n");
         scanf("%d",&choice);
         if (choice==0)
         {
            printf("\nGoodbye\n");
            break;
         }

         switch (choice)
         {
         case 1:
         {
            printf("Enter number to insert: ");
            scanf("%d", &num);
            insert_quad(h,num);
            printf("\n");
            break;
         }
         case 2:
         {
            print_hash(h);
            break;
         }

         default:
         {
            printf("Error");
            break;
         }
         }
      }
      break;

   }
      case 3:
   {
      printf("Welcome to Double Hashing \n Enter 1 to insert \n Enter 2
to print the entire Table \n Enter 0 to Exit \n");
      while (true)
```

```
        {
           int choice=0;
           printf("Enter choice \n");
           scanf("%d",&choice);
           if (choice==0)
           {
              printf("\nGoodbye\n");
              break;
           }

           switch (choice)
           {
           case 1:
           {
              printf("Enter number to insert: ");
              scanf("%d", &num);
              insert_doub(h,num);
              printf("\n");
              break;
           }
           case 2:
           {
              print_hash(h);
              break;
           }

           default:
           {
              printf("Error");
              break;
           }
           }
        }
        break;

     }

     default:
     {
        printf("Error");
        break;
     }


     }

}

int main(int argc, char const *argv[])
{
```

```
    hash * h = (hash * ) malloc(sizeof(hash));

    printf("Enter size of the hash table: ");
    int hash_size=10;
    scanf("%d",&hash_size);

    h->size=hash_size;
    h->arr=(int *)malloc(h->size *sizeof(int));
    h->status=(char *)malloc(h->size *sizeof(char));
    for (int i = 0; i < h->size; i++)
    {
       h->status[i]='E';
       h->arr[i]=-1;
    }

    choice(h);

    free(h->arr);
    free(h->status);
    free(h);
    return 0;
}
```

Output

LINEAR

```
Enter size of the hash table: 10
Enter 1 to Do linear probing
Enter 2 to do Quadratic probing:
Enter 3 to do Double Hashing
1
Welcome to linear probing
 Enter 1 to insert
 Enter 2 to print the entire Table
 Enter 0 to Exit
Enter choice
1
Enter number to insert: 77

 Applying Probe 0: (77 % 10) % 10 = 7
Element 77 has been inserted at 7
Enter choice
1
Enter number to insert: 153

 Applying Probe 0: (153 % 10) % 10 =
Element 153 has been inserted at 3
Enter choice
1
Enter number to insert: 1

 Applying Probe 0: (1 % 10) % 10 = 1
Element 1 has been inserted at 1
Enter choice
1
Enter number to insert: 3
```

```
 Applying Probe 0: (3 % 10) % 10 = 3
A Collision has occured:
Applying Probe 1: (3+1) % 10 = 4
Element 3 has been inserted at 4
Enter choice
1
Enter number to insert: 79

 Applying Probe 0: (79 % 10) % 10 = 9
Element 79 has been inserted at 9
Enter choice
1
Enter number to insert: 155

 Applying Probe 0: (155 % 10) % 10 = 5
Element 155 has been inserted at 5
Enter choice
1
Enter number to insert: 0

 Applying Probe 0: (0 % 10) % 10 = 0
Element 0 has been inserted at 0
Enter choice
1
Enter number to insert: 152

 Applying Probe 0: (152 % 10) % 10 = 2
Element 152 has been inserted at 2
Enter choice
1
Enter number to insert: 152
```

```
 Applying Probe 0: (152 % 10) % 10 = 2
You Have a Duplicate: No need to insert:
Enter choice
1
Enter number to insert: 71

 Applying Probe 0: (71 % 10) % 10 = 1
A Collision has occured:
Applying Probe 1: (71+1) % 10 = 2
Applying Probe 2: (71+2) % 10 = 4
Applying Probe 3: (71+3) % 10 = 7
Applying Probe 4: (71+4) % 10 = 1
Applying Probe 5: (71+5) % 10 = 6
Element 71 has been inserted at 6
Enter choice
2

 Index  Element Stored  Status
0           0            F
1           1            F
2           152          F
3           153          F
4           3            F
5           155          F
6           71           F
7           77           F
8           -1           E
9           79           F
Enter choice
0

Goodbye
```

Quadratic

```
Enter size of the hash table: 19
Enter 1 to Do linear probing
Enter 2 to do Quadratic probing:
Enter 3 to do Double Hashing
2
Welcome to Quadratic probing
 Enter 1 to insert
 Enter 2 to print the entire Table
 Enter 0 to Exit
Enter choice
1
Enter number to insert: 86

 Applying Probe 0: (86 % 19) % 19 = 10
Element 86 has been inserted at 10

Enter choice
1
Enter number to insert: 198

 Applying Probe 0: (198 % 19) % 19 = 8
Element 198 has been inserted at 8

Enter choice
1
Enter number to insert: 466

  Applying Probe 0: (466 % 19) % 19 = 10
 A Collision has occured:
 Applying Probe 1: ( 10 + 1 ) % 19 = 11
 Element 466 has been inserted at 11

 Enter choice
 1
 Enter number to insert: 709

  Applying Probe 0: (709 % 19) % 19 = 6
 Element 709 has been inserted at 6

 Enter choice
 1
 Enter number to insert: 973

  Applying Probe 0: (973 % 19) % 19 = 4
 Element 973 has been inserted at 4

 Enter choice
 1
 Enter number to insert: 981

  Applying Probe 0: (981 % 19) % 19 = 12
 Element 981 has been inserted at 12

 Enter choice
 1
 Enter number to insert: 374
```

```
 Applying Probe 0: (374 % 19) % 19 = 13
Element 374 has been inserted at 13

Enter choice
1
Enter number to insert: 766

 Applying Probe 0: (766 % 19) % 19 = 6
A Collision has occured:
Applying Probe 1: ( 6 + 1 ) % 19 = 7
Element 766 has been inserted at 7

Enter choice
1
Enter number to insert: 473

 Applying Probe 0: (473 % 19) % 19 = 17
Element 473 has been inserted at 17

Enter choice
1
Enter number to insert: 198
You Have a Duplicate: No need to insert:

Enter choice
1
Enter number to insert: 342
```

```
 Applying Probe 0: (342 % 19) % 19 = 0
Element 342 has been inserted at 0

Enter choice
1
Enter number to insert: 191

 Applying Probe 0: (191 % 19) % 19 = 1
Element 191 has been inserted at 1

Enter choice
1
Enter number to insert: 393

 Applying Probe 0: (393 % 19) % 19 = 13
A Collision has occured:
Applying Probe 1: ( 13 + 1 ) % 19 = 14
Element 393 has been inserted at 14

Enter choice
1
Enter number to insert: 300

 Applying Probe 0: (300 % 19) % 19 = 15
Element 300 has been inserted at 15

Enter choice
1
Enter number to insert: 11

 Applying Probe 0: (11 % 19) % 19 = 11
A Collision has occured:
Applying Probe 1: ( 11 + 1 ) % 19 = 12
Applying Probe 2: ( 11 + 4 ) % 19 = 16
Element 11 has been inserted at 16

Enter choice
1
Enter number to insert: 538

 Applying Probe 0: (538 % 19) % 19 = 6
A Collision has occured:
Applying Probe 1: ( 6 + 1 ) % 19 = 7
Applying Probe 2: ( 6 + 4 ) % 19 = 11
Applying Probe 3: ( 6 + 9 ) % 19 = 1
Applying Probe 4: ( 6 + 16 ) % 19 = 17
Applying Probe 5: ( 6 + 25 ) % 19 = 4
Applying Probe 6: ( 6 + 36 ) % 19 = 2
Element 538 has been inserted at 2

Enter choice
1
Enter number to insert: 913
```

```
 Applying Probe 0: (913 % 19) % 19 = 1
A Collision has occured:
Applying Probe 1: ( 1 + 1 ) % 19 = 2
Applying Probe 2: ( 1 + 4 ) % 19 = 6
Applying Probe 3: ( 1 + 9 ) % 19 = 15
Applying Probe 4: ( 1 + 16 ) % 19 = 12
Applying Probe 5: ( 1 + 25 ) % 19 = 18
Element 913 has been inserted at 18

Enter choice
1
Enter number to insert: 220

 Applying Probe 0: (220 % 19) % 19 = 11
A Collision has occured:
Applying Probe 1: ( 11 + 1 ) % 19 = 12
Applying Probe 2: ( 11 + 4 ) % 19 = 16
Applying Probe 3: ( 11 + 9 ) % 19 = 6
Applying Probe 4: ( 11 + 16 ) % 19 = 3
Element 220 has been inserted at 3

Enter choice
1
Enter number to insert: 844

 Applying Probe 0: (844 % 19) % 19 = 8
A Collision has occured:
Applying Probe 1: ( 8 + 1 ) % 19 = 9
Element 844 has been inserted at 9
Enter choice
1
Enter number to insert: 565

 Applying Probe 0: (565 % 19) % 19 = 14
A Collision has occured:
Applying Probe 1: ( 14 + 1 ) % 19 = 15
Applying Probe 2: ( 14 + 4 ) % 19 = 0
Applying Probe 3: ( 14 + 9 ) % 19 = 9
Applying Probe 4: ( 14 + 16 ) % 19 = 6
Applying Probe 5: ( 14 + 25 ) % 19 = 12
Applying Probe 6: ( 14 + 36 ) % 19 = 10
Applying Probe 7: ( 14 + 49 ) % 19 = 2
Applying Probe 8: ( 14 + 64 ) % 19 = 9
Applying Probe 9: ( 14 + 81 ) % 19 = 14
Applying Probe 10: ( 14 + 100 ) % 19 = 0
Applying Probe 11: ( 14 + 121 ) % 19 = 7
Applying Probe 12: ( 14 + 144 ) % 19 = 18
Applying Probe 13: ( 14 + 169 ) % 19 = 16
Applying Probe 14: ( 14 + 196 ) % 19 = 3
Applying Probe 15: ( 14 + 225 ) % 19 = 0
Applying Probe 16: ( 14 + 256 ) % 19 = 9
Applying Probe 17: ( 14 + 289 ) % 19 = 13
Applying Probe 18: ( 14 + 324 ) % 19 = 14
Applying Probe 19: ( 14 + 361 ) % 19 = 14
Applying Probe 20: ( 14 + 400 ) % 19 = 15
Applying Probe 21: ( 14 + 441 ) % 19 = 0
Applying Probe 22: ( 14 + 484 ) % 19 = 9
Applying Probe 23: ( 14 + 529 ) % 19 = 6
```

```
Applying Probe 24: ( 14 + 576 ) % 19 = 12
Applying Probe 25: ( 14 + 625 ) % 19 = 10
Applying Probe 26: ( 14 + 676 ) % 19 = 2
Applying Probe 27: ( 14 + 729 ) % 19 = 9
Applying Probe 28: ( 14 + 784 ) % 19 = 14
Applying Probe 29: ( 14 + 841 ) % 19 = 0
Applying Probe 30: ( 14 + 900 ) % 19 = 7
Applying Probe 31: ( 14 + 961 ) % 19 = 18
Applying Probe 32: ( 14 + 1024 ) % 19 = 16
Applying Probe 33: ( 14 + 1089 ) % 19 = 3
Applying Probe 34: ( 14 + 1156 ) % 19 = 0
Applying Probe 35: ( 14 + 1225 ) % 19 = 9
Applying Probe 36: ( 14 + 1296 ) % 19 = 13
Applying Probe 37: ( 14 + 1369 ) % 19 = 14
Applying Probe 38: ( 14 + 1444 ) % 19 = 14
Element 565 is going in infinite loop:
Enter choice
2

 Index  Element Stored  Status
0           342          F
1           191          F
2           538          F
3           220          F
4           973          F
5           -1           E
6           709          F
7           766          F
8           198          F
9           844          F
10          86           F
11          466          F
12          981          F
13          374          F
14          393          F
15          300          F
16          11           F
17          473          F
18          913          F
Enter choice
0

Goodbye
```

Double

```
Enter size of the hash table: 10
Enter 1 to Do linear probing
Enter 2 to do Quadratic probing:
Enter 3 to do Double Hashing
3
Welcome to Double Hashing
 Enter 1 to insert
 Enter 2 to print the entire Table
 Enter 0 to Exit
Enter choice
1
Enter number to insert: 77

 Applying Probe 0: (77 % 10) % 10 = 7
Element 77 has been inserted at 7

Enter choice
1
Enter number to insert: 153

 Applying Probe 0: (153 % 10) % 10 = 3
Element 153 has been inserted at 3

Enter choice
1
Enter number to insert: 1

 Applying Probe 0: (1 % 10) % 10 = 1
Element 1 has been inserted at 1
```

```
Enter choice
1
Enter number to insert: 3

 Applying Probe 0: (3 % 10) % 10 = 3
A Collision has occured:
Applying Probe 1: ( 3 + 1*4 ) % 10 = 7
Applying Probe 2: ( 3 + 2*4 ) % 10 = 1
Applying Probe 3: ( 3 + 3*4 ) % 10 = 5
Element 3 has been inserted at 5

Enter choice
1
Enter number to insert: 79

 Applying Probe 0: (79 % 10) % 10 = 9
Element 79 has been inserted at 9

Enter choice
1
Enter number to insert: 155

 Applying Probe 0: (155 % 10) % 10 = 5
A Collision has occured:
Applying Probe 1: ( 5 + 1*7 ) % 10 = 2
Element 155 has been inserted at 2

Enter choice
1
Enter number to insert: 0

 Applying Probe 0: (0 % 10) % 10 = 0
Element 0 has been inserted at 0

Enter choice
1
Enter number to insert: 76

 Applying Probe 0: (76 % 10) % 10 = 6
Element 76 has been inserted at 6
```

```
Enter choice
1
Enter number to insert: 152

 Applying Probe 0: (152 % 10) % 10 = 2
A Collision has occured:
Applying Probe 1: ( 2 + 1*5 ) % 10 = 7
Applying Probe 2: ( 2 + 2*5 ) % 10 = 2
Applying Probe 3: ( 2 + 3*5 ) % 10 = 7
Applying Probe 4: ( 2 + 4*5 ) % 10 = 2
Applying Probe 5: ( 2 + 5*5 ) % 10 = 7
Applying Probe 6: ( 2 + 6*5 ) % 10 = 2
Applying Probe 7: ( 2 + 7*5 ) % 10 = 7
Applying Probe 8: ( 2 + 8*5 ) % 10 = 2
Applying Probe 9: ( 2 + 9*5 ) % 10 = 7
Applying Probe 10: ( 2 + 10*5 ) % 10 = 2
Applying Probe 11: ( 2 + 11*5 ) % 10 = 7
Applying Probe 12: ( 2 + 12*5 ) % 10 = 2
Applying Probe 13: ( 2 + 13*5 ) % 10 = 7
Applying Probe 14: ( 2 + 14*5 ) % 10 = 2
Applying Probe 15: ( 2 + 15*5 ) % 10 = 7
Applying Probe 16: ( 2 + 16*5 ) % 10 = 2
Applying Probe 17: ( 2 + 17*5 ) % 10 = 7
Applying Probe 18: ( 2 + 18*5 ) % 10 = 2
Applying Probe 19: ( 2 + 19*5 ) % 10 = 7
Applying Probe 20: ( 2 + 20*5 ) % 10 = 2
Element 152 is going in infinite loop:

Enter choice
1
Enter number to insert: 152

 Applying Probe 0: (152 % 10) % 10 = 2
A Collision has occured:
Applying Probe 1: ( 2 + 1*5 ) % 10 = 7
Applying Probe 2: ( 2 + 2*5 ) % 10 = 2
Applying Probe 3: ( 2 + 3*5 ) % 10 = 7
Applying Probe 4: ( 2 + 4*5 ) % 10 = 2
Applying Probe 5: ( 2 + 5*5 ) % 10 = 7
Applying Probe 6: ( 2 + 6*5 ) % 10 = 2
Applying Probe 7: ( 2 + 7*5 ) % 10 = 7
Applying Probe 8: ( 2 + 8*5 ) % 10 = 2
Applying Probe 9: ( 2 + 9*5 ) % 10 = 7
Applying Probe 10: ( 2 + 10*5 ) % 10 = 2
Applying Probe 11: ( 2 + 11*5 ) % 10 = 7
Applying Probe 12: ( 2 + 12*5 ) % 10 = 2
Applying Probe 13: ( 2 + 13*5 ) % 10 = 7
Applying Probe 14: ( 2 + 14*5 ) % 10 = 2
Applying Probe 15: ( 2 + 15*5 ) % 10 = 7
Applying Probe 16: ( 2 + 16*5 ) % 10 = 2
Applying Probe 17: ( 2 + 17*5 ) % 10 = 7
Applying Probe 18: ( 2 + 18*5 ) % 10 = 2
Applying Probe 19: ( 2 + 19*5 ) % 10 = 7
Applying Probe 20: ( 2 + 20*5 ) % 10 = 2
Element 152 is going in infinite loop:
```

<table>
<tr><td></td><td>

```
Enter choice
2

 Index  Element Stored  Status
0           0            F
1           1            F
2           155          F
3           153          F
4           -1           E
5           3            F
6           76           F
7           77           F
8           -1           E
9           79           F
Enter choice
0

Goodbye
```

</td></tr>
<tr><td>Conclusion</td><td>Thus, we implemented a hash table that employs three collision resolution techniques: linear probing, quadratic probing, and double hashing. Each method effectively manages collisions, but with different performance implications. Linear probing is simpler but may lead to clustering, while quadratic probing mitigates this issue through better distribution. Double hashing offers an efficient alternative by utilizing a secondary hash function.</td></tr>
</table>