Aim: Hashing Techniques.

Objectives: The main objective of this assignment is to understand and implement various hashing techniques such as modulo division, mid-square, digit extraction, fold shift and fold boundary in C++. The goal is to understand how different hashing algorithms can be used to store the data in a specific location.

Tools Used: VS Code C++.

Concept:

Linear Probing is a collision resolution technique where, if the calculated index is already occupied, the next available index is checked by incrementing the current index in a sequential manner.

Modulo Division

Modulo Division, as a hash function, works by taking a key (usually a number) and dividing it by the size of the hash table. The division gives a remainder, which is used as the index where the key will be stored in the table.

Algorithm:

1) For each element, calculate the hash value using: hash = k (key)% n (Size of the table).
2) If the hash index is free, insert the key; otherwise, apply linear probing.

Example:

Consider a table of size 7 (n = 7) and the keys: 10, 22, 31.

- Hash (10) = 10 % 7 = 3 (insert at index 3).
- Hash (22) = 22 % 7 = 1 (insert at index 1).
- Hash (31) = 31 % 7 = 3 (collision at index 3, so check next index 4). Insert at index 4.

Mid-Square

Mid-Square Method, The Mid-Square hash function works by squaring the key and extracting the middle digits of the result to determine the index in the hash table.

Algorithm:

1. For each element, calculate the hash value by first squaring the key: prod = key * key.
2. Extract the middle digits from the squared result.
3. Then, hash index = Middle digit % size of the table
4. .

5. If the hash index (determined by the middle digits) is free, insert the key; otherwise, apply linear probing.

Example:
Consider a table of size 7 (n = 7) and the keys: 12, 23, 34.

- Hash (12):
  Square the key: 12 * 12 = 144.
  Extract the middle digits: 4.
  Insert the key at index 4.
- Hash (23):
  Square the key: 23 * 23 = 529.
  Extract the middle digit: 2.
  Insert the key at index 2.
- Hash (34):
  Square the key: 34 * 34 = 1156.
  Extract the middle digits: 15.
  Insert the key at index 1.

Digit Extraction Method

The Digit Extraction hash function extracts specific digits from the key based on user-defined positions and uses those digits to determine the index in the hash table.

Algorithm:

1. For each key, the user specifies which digits from the key should be extracted to form the hash value.
2. Extract the digits from the key based on the specified positions.
3. Then, hash index = Middle digit % size of the table.
4. If the hash index is free, insert the key; otherwise, apply linear probing to resolve collisions.

Example:

Consider a hash table of size 10 (n = 10) and the keys: 1234, 5678, and 91011. Assume the digits to be extracted are at positions: 1 (units place) and 3 (hundreds place).

- Hash (1234):
  Extract digits at positions 1 and 3 from 1234 → Digits: 4 and 2 → Hash value: 24 → Insert at index 24 % 10 = 4.
- Hash (5678):
  Extract digits at positions 1 and 3 from 5678 → Digits: 8 and 6 → Hash value: 68 → Insert at index 68 % 10 = 8.
- Hash (91011):
  Extract digits at positions 1 and 3 from 91011 → Digits: 1 and 0 → Hash value: 01 → Insert at index 01 % 10 = 1.

Fold Shift Method

The Fold Shift hash function works by dividing the key into smaller parts (chunks) and adding them together. The resulting sum is then used to generate the hash value.

Algorithm:

1. Break the key into smaller parts based on the length of the hash table's size.
2. Add all the smaller parts together to form a new value and ignore the carry if any.
3. Take the remainder of the sum when divided by the size of the table to get the hash index.
4. If the hash index is free, insert the key; otherwise, apply linear probing to resolve collisions.

Example:

Consider a hash table of size 10 (n = 10) and the keys: 123456, 789123.

- Hash (123456):
  Break the key into chunks of 2 digits: (12, 34, 56).
  Sum of chunks: 12 + 34 + 56 = 102.
  Hash value: 102 % 10 = 2 → Insert at index 2.
- Hash (789123):
  Break the key into chunks of 2 digits: (78, 91, 23).
  Sum of chunks: 78 + 91 + 23 = 192.
  Hash value: 192 % 10 = 2.
  Since index 2 is already occupied, apply linear probing and insert at the next available index, which is 3.

Fold Boundary Method

The Fold Boundary hash function is a variation of the fold method where the key is divided into parts, and some of the parts are reversed (typically the first and last parts).

Algorithm**:**

1. Break the key into smaller parts based on the length of the hash table's size.
2. Reverse the first and last parts of the key.
3. Add all the parts together to form a new value and ignore the carry if any.
4. Take the remainder of the sum when divided by the size of the table to get the hash index.
5. If the hash index is free, insert the key; otherwise, apply linear probing to resolve collisions.

Example:

Consider a hash table of size 10 (n = 10) and the keys: 123456, 789123.

- Hash (123456):

  Break the key into chunks of 2 digits: (12, 34, 56).

  Reverse the first and last parts: (21, 34, 65).

  Sum of chunks: 21 + 34 + 65 = 120.

Igone the carry -> 1.

Hash value: 20 % 10 = 0 → Insert at index 0.

- Hash (789123):

  Break the key into chunks of 2 digits: (78, 91, 23).

  Reverse the first and last parts: (87, 91, 32).

  Sum of chunks: 87 + 91 + 32 = 210.

  Igone the carry -> 2.

  Hash value: 10 % 10 = 0.

  Since index 0 is already occupied, apply linear probing and insert at the next available index, which is 1.

Problem Statement:

1.) Implement modulo division hash function with linear probing.

2.) Implement mid-square hash function with linear probing.

3.) Implement digit extraction hash function with linear probing.

4.) Implement fold shift hash function with linear probing.

5.) Implement fold boundary hash function with linear probing.

Solution:

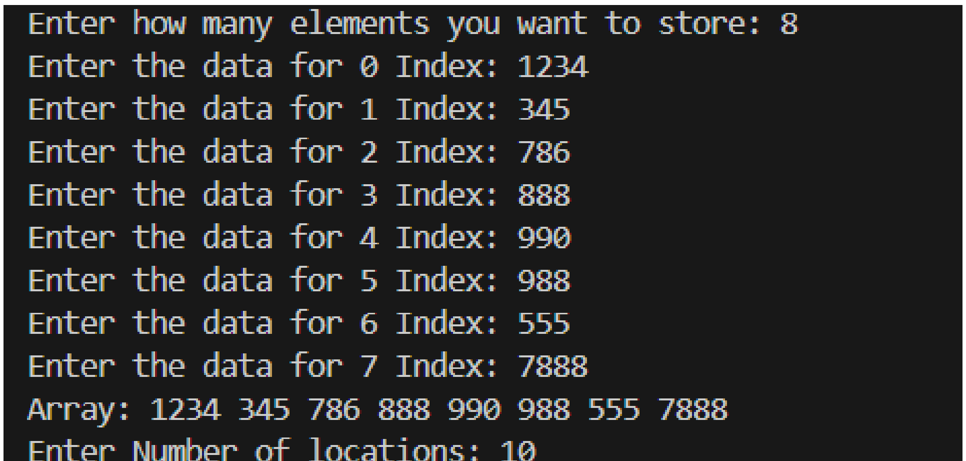
1) Modulo division hash function with linear probing.

```
#include <iostream>

using namespace std;

#define max 100


int n, m, arr[max], arrM[max], collision = 0;
```

```
bool err = false;

class ModuloDivision
{
public:
  void input()
  {
    cout << "Enter how many elements you want to store: ";
    cin >> m;
    for (int i = 0; i < m; i++)
    {
      cout << "Enter the data for " << i << " Index: ";
      cin >> arr[i];
    }
    cout << "Array: ";
    for (int i = 0; i < m; i++)
    {
      cout << arr[i] << " ";
    }
    cout << endl;
    cout << "Enter Number of locations: ";
    cin >> n;

    if (n < m)
    {
      cout << "Size of Hash Table must br greater than or equal to the no. of elements";
      err = true;
      return;
    }

    arrM[n] = {0};
  }

  void placing()
```

```
{
    for (int i = 0; i < m; i++)
    {
        int hk = mdivision(n, arr[i]);
        linearProbing(hk, i);
    }
}

int mdivision(int n, int k)
{
    int h;
    h = k % n;
    return h;
}

void linearProbing(int hk, int i)
{
    if (hk < n)
    {


        if (arrM[hk] == 0)
        {
            arrM[hk] = arr[i];
        }
        else
        {
            hk++;
            collision++;
            linearProbing(hk, i);
        }
    }
    else
    {
        hk = hk % n;
```

```
            linearProbing(hk, i);
        }
    }
    void output()
    {
        cout << "No. of Collision: " << collision << endl;
        cout << "Value | Index " << endl;
        for (int i = 0; i < n; i++)
        {
            if (arrM[i] != 0)
            {
                cout << arrM[i] << " | " << i << endl;
            }
        }
        cout << endl;
    }
};

int main()
{
    ModuloDivision md;
    md.input();
    if (!err)
    {
        md.placing();
        md.output();
    }
    return 0;
}
```



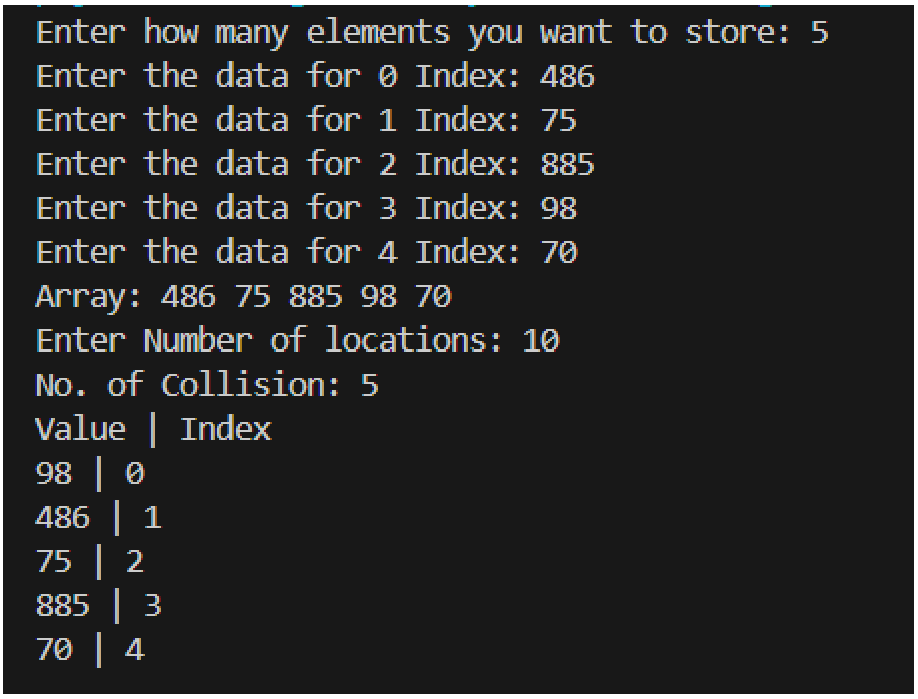
```
Enter how many elements you want to store: 8
Enter the data for 0 Index: 1234
Enter the data for 1 Index: 345
Enter the data for 2 Index: 786
Enter the data for 3 Index: 888
Enter the data for 4 Index: 990
Enter the data for 5 Index: 988
Enter the data for 6 Index: 555
Enter the data for 7 Index: 7888
Array: 1234 345 786 888 990 988 555 7888
Enter Number of locations: 10
```

2) Mid-square hash function with linear probing

```
#include <iostream>
#include <string>
using namespace std;
#define max 100

int n, m, arr[max], arrM[max], collision = 0;
bool err = false;

class Midsquare
{
public:
    void input()
    {
        cout << "Enter how many elements you want to store: ";
        cin >> m;
        for (int i = 0; i < m; i++)
        {
            cout << "Enter the data for " << i << " Index: ";
            cin >> arr[i];
        }
        cout << "Array: ";
        for (int i = 0; i < m; i++)
        {
            cout << arr[i] << " ";
        }
        cout << endl;
        cout << "Enter Number of locations: ";
        cin >> n;

        if (n < m)
        {
            cout << "Size of Hash Table must br greater than or equal to the no. of elements";
            err = true;
            return;
        }
        arrM[n] = {0};
    }
    void placing()
    {
        for (int i = 0; i < m; i++)
        {
            int hk = msquare(n, arr[i]);
            linearProbing(hk, i);
        }
    }

    int msquare(int n, int k)
    {
        int prod = k * k;
        string pd = to_string(prod);
        int length = pd.length();
        string val = "";
        int value = 0;

        int mid = length / 2;

        if (length % 2 == 0)
        {
            val = string(1, pd.at(mid - 1)) + string(1, pd.at(mid));
        }
        else
        {
            val = pd.at(mid);
        }

        value = stoi(val);
```

```
        return value;
    }

    void linearProbing(int hk, int i)
    {
        if (hk < n)
        {

            if (arrM[hk] == 0)
            {
                arrM[hk] = arr[i];
            }
            else
            {
                hk++;
                collision++;
                linearProbing(hk, i);
            }
        }
        else
        {
            hk = hk % n;
            linearProbing(hk, i);
        }
    }

    void output()
    {
        cout << "No. of Collision: " << collision << endl;
        cout << "Value | Index " << endl;
        for (int i = 0; i < n; i++)
        {
            if (arrM[i] != 0)
            {
```

```
            cout << arrM[i] << " | " << i << endl;
          }
        }
        cout << endl;
    }
};
int main()
{
    Midsquare ms;
    ms.input();
    if (!err)
    {
        ms.placing();
        ms.output();
    }
    return 0;
}
```



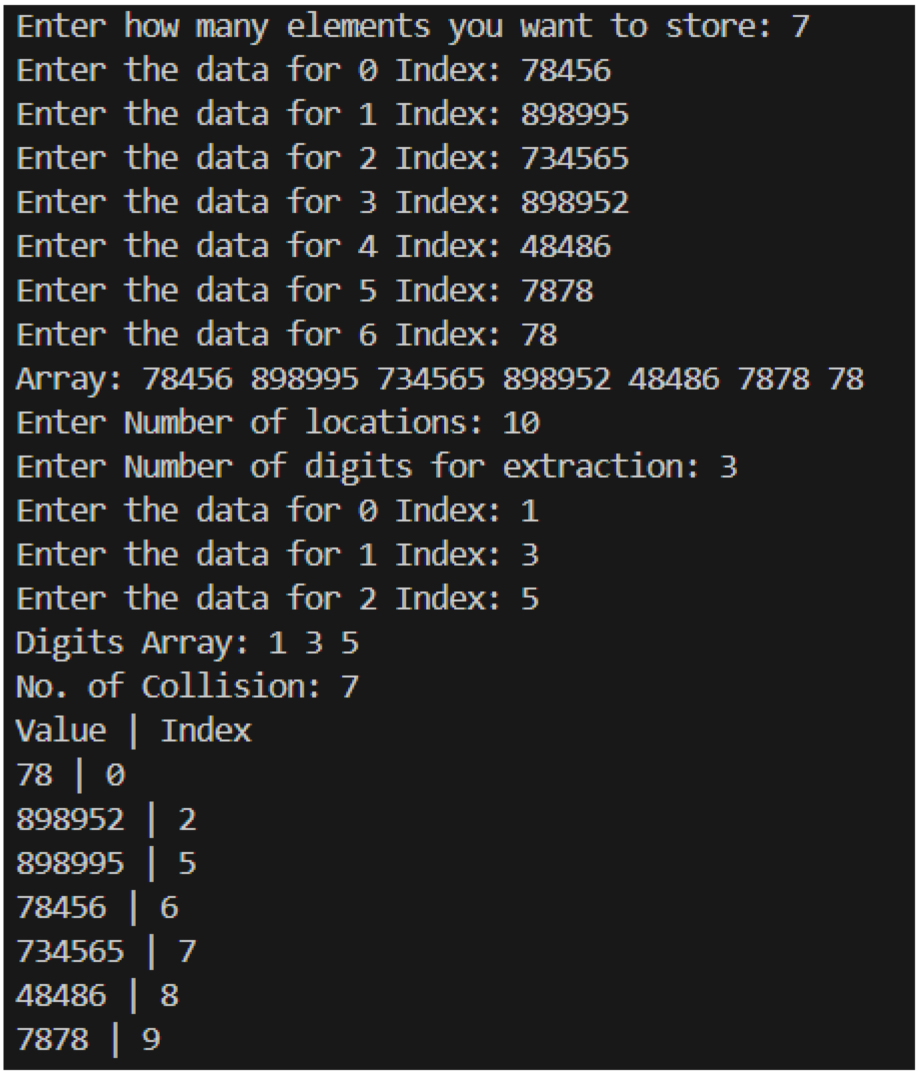
```
Enter how many elements you want to store: 5
Enter the data for 0 Index: 486
Enter the data for 1 Index: 75
Enter the data for 2 Index: 885
Enter the data for 3 Index: 98
Enter the data for 4 Index: 70
Array: 486 75 885 98 70
Enter Number of locations: 10
No. of Collision: 5
Value | Index
98 | 0
486 | 1
75 | 2
885 | 3
70 | 4
```

3) Digit Extraction hash function with linear probing

```
#include <iostream>
#include <algorithm>

using namespace std;
#define max 100

int n, m, arr[max], arrM[max], num, digits[max], collision = 0;
bool err = false;

class DigitExtraction
{
public:
        void input()
        {
                cout << "Enter how many elements you want to store: ";
                cin >> m;
                for (int i = 0; i < m; i++)
                {
                        cout << "Enter the data for " << i << " Index: ";
                        cin >> arr[i];
                }
                cout << "Array: ";
                for (int i = 0; i < m; i++)
                {
                        cout << arr[i] << " ";
                }
                cout << endl;
                cout << "Enter Number of locations: ";
                cin >> n;

                if (n < m)
```

```
		{
			cout << "Size of Hash Table must br greater than or equal to the no. of
elements";
			err = true;
			return;
		}

		arrM[n] = {0};

		cout << "Enter Number of digits for extraction: ";
		cin >> num;

		for (int i = 0; i < num; i++)
		{
			cout << "Enter the data for " << i << " Index: ";
			cin >> digits[i];
		}

		cout << "Digits Array: ";
		for (int i = 0; i < num; i++)
		{
			cout << digits[i] << " ";
		}
		cout << endl;
	}

	void placing()
	{
		for (int i = 0; i < m; i++)
		{
			int hk = dextraction(n, arr[i]);
			linearProbing(hk, i);
		}
	}
```

```
int dextraction(int n, int k)
{
        string val = "";
        int value;
        string key = to_string(k);

        int j = 0;
        int count = 1;

        for (int i = key.length() - 1; i >= 0; i--)
        {
                if (digits[j] == count)
                {
                        val.push_back(key.at(i));
                        j++;
                }
                count++;
        }
        reverse(val.begin(), val.end());
        value = stoi(val);

        return value;
}

void linearProbing(int hk, int i)
{
        if (hk < n)
        {

                if (arrM[hk] == 0)
                {
                        arrM[hk] = arr[i];
                }
                else
                {
                    hk++;
                    collision++;
                    linearProbing(hk, i);
                }
            }
            else
            {
                hk = hk % n;
                linearProbing(hk, i);
            }
        }
        void output()
        {
            cout << "No. of Collision: " << collision << endl;
            cout << "Value | Index " << endl;
            for (int i = 0; i < n; i++)
            {
                if (arrM[i] != 0)
                {
                    cout << arrM[i] << " | " << i << endl;
                }
            }
            cout << endl;
        }
};

int main()
{
        DigitExtraction de;
        de.input();
        if (!err)
        {
            de.placing();
            de.output();
        }
        return 0;
}
```



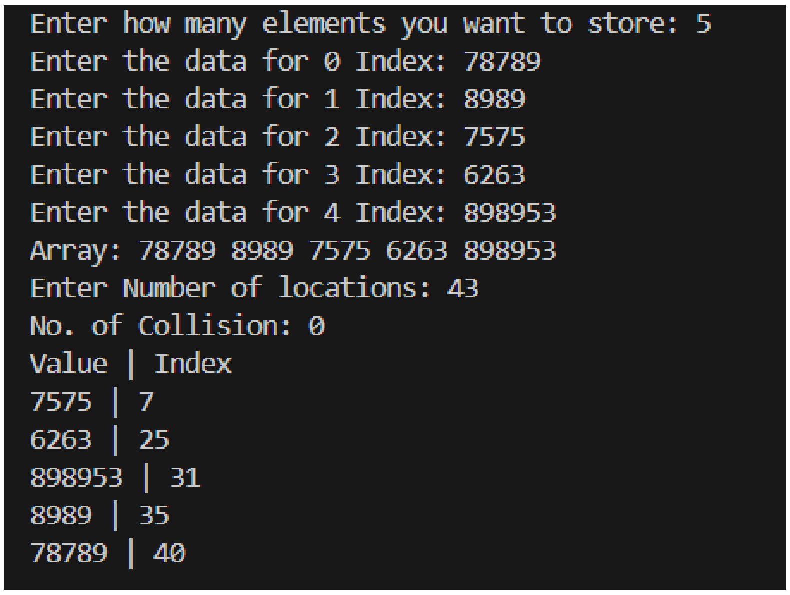
```
Enter how many elements you want to store: 7
Enter the data for 0 Index: 78456
Enter the data for 1 Index: 898995
Enter the data for 2 Index: 734565
Enter the data for 3 Index: 898952
Enter the data for 4 Index: 48486
Enter the data for 5 Index: 7878
Enter the data for 6 Index: 78
Array: 78456 898995 734565 898952 48486 7878 78
Enter Number of locations: 10
Enter Number of digits for extraction: 3
Enter the data for 0 Index: 1
Enter the data for 1 Index: 3
Enter the data for 2 Index: 5
Digits Array: 1 3 5
No. of Collision: 7
Value | Index
78 | 0
898952 | 2
898995 | 5
78456 | 6
734565 | 7
48486 | 8
7878 | 9
```

4) Fold Shift hash function with linear probing

```
#include <iostream>
#include <algorithm>
#include <string>

using namespace std;
#define MAX_LEN 100
int n, m, arr[MAX_LEN], arrM[MAX_LEN], num, digits[MAX_LEN], collision = 0;
bool err = false;

class FoldShift
{
public:
        void input()
        {
                cout << "Enter how many elements you want to store: ";
                cin >> m;
                for (int i = 0; i < m; i++)
                {
                        cout << "Enter the data for " << i << " Index: ";
                        cin >> arr[i];
                }
                cout << "Array: ";
                for (int i = 0; i < m; i++)
                {
                        cout << arr[i] << " ";
                }
                cout << endl;
                cout << "Enter Number of locations: ";
                cin >> n;
                if (n < m)
                {
```

```
                cout << "Size of Hash Table must br greater than or equal to the no. of
elements";
                err = true;
                return;
            }
            arrM[n] = {0};
    }

    void placing()
    {
            for (int i = 0; i < m; i++)
            {
                int hk = fshift(n, arr[i]);
                linearProbing(hk, i);
            }
    }

    int fshift(int n, int k)
    {
            int value = 0;

            int len = to_string(n).length();

            string str = to_string(k);

            for (int i = str.length(); i > 0; i -= len)
            {
                int start = max(0, i - len);
                string part = str.substr(start, i - start);

                value += stoi(part);

                int max_value = 1;
                for (int j = 0; j < len; j++)
                {
                        max_value *= 10; // Multiply by 10 'len' times to get 10^len
                }
                // Ensure that value is restricted to last 'len' digits
                value = value % max_value;
            }
            // cout << "value" << value;
            return value;
    }

    void linearProbing(int hk, int i)
    {
            if (hk < n)
            {

                if (arrM[hk] == 0)
                {
                        arrM[hk] = arr[i];
                }
                else
                {
                        hk++;
                        collision++;
                        linearProbing(hk, i);
                }
            }
            else
            {
                hk = hk % n;
                linearProbing(hk, i);
            }
    }

    void output()
```

```
        {
                cout << "No. of Collision: " << collision << endl;
                cout << "Value | Index " << endl;
                for (int i = 0; i < n; i++)
                {
                        if (arrM[i] != 0)
                        {
                                cout << arrM[i] << " | " << i << endl;
                        }
                }
                cout << endl;
        }
};

int main()
{
        FoldShift fs;
        fs.input();
        if (!err)
        {
                fs.placing();
                fs.output();
        }
        return 0;
}
```

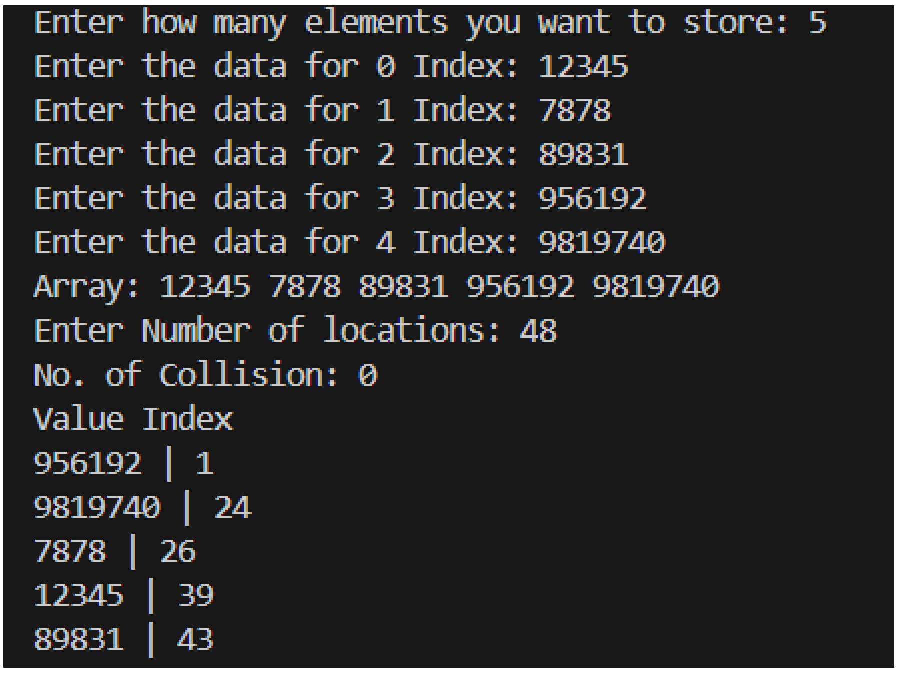


```
Enter how many elements you want to store: 5
Enter the data for 0 Index: 78789
Enter the data for 1 Index: 8989
Enter the data for 2 Index: 7575
Enter the data for 3 Index: 6263
Enter the data for 4 Index: 898953
Array: 78789 8989 7575 6263 898953
Enter Number of locations: 43
No. of Collision: 0
Value | Index
7575 | 7
6263 | 25
898953 | 31
8989 | 35
78789 | 40
```

5) Fold Boundary hash function with linear probing

```
#include <iostream>
#include <algorithm>
#include <string>

using namespace std;
#define MAX_LEN 100

int n, m, arr[MAX_LEN], arrM[MAX_LEN], num, digits[MAX_LEN], collision = 0;
bool err = false;

class FoldBoundary
{
public:
        void input()
        {
                cout << "Enter how many elements you want to store: ";
                cin >> m;
                for (int i = 0; i < m; i++)
                {
                        cout << "Enter the data for " << i << " Index: ";
                        cin >> arr[i];
                }
                cout << "Array: ";
                for (int i = 0; i < m; i++)
                {
                        cout << arr[i] << " ";
                }
                cout << endl;
                cout << "Enter Number of locations: ";
                cin >> n;

                if (n < m)
                {
                        cout << "Size of Hash Table must br greater than or equal to the no. of
elements";
                        err = true;
                        return;
                }

                arrM[n] = {0};
        }

        void placing()
        {
                for (int i = 0; i < m; i++)
                {
                        int hk = fshift(n, arr[i]);
                        linearProbing(hk, i);
                }
        }

        int fshift(int n, int k)
        {
                int value = 0;

                int len = to_string(n).length();

                string str = to_string(k);

                for (int i = str.length(); i > 0; i -= len)
                {
                        int start = max(0, i - len);
                        string part = str.substr(start, i - start);

                        while (part.length() < len)
                        {
                            part = "0" + part;
                        }

                        if (i == str.length() || start == 0)
                        {
                            reverse(part.begin(), part.end());
                        }

                        // cout << "part :" << i << " "<< part << endl;

                        value += stoi(part);

                        int max_value = 1;
                        for (int j = 0; j < len; j++)
                        {
                            max_value *= 10; // Multiply by 10 'len' times to get 10^len
                        }
                        // Ensure that value is restricted to last 'len' digits
                        value = value % max_value;
                }
                return value;
        }

        void linearProbing(int hk, int i)
        {
                if (hk < n)
                {

                        if (arrM[hk] == 0)
                        {
                            arrM[hk] = arr[i];
                        }
                        else
                        {
                            hk++;
                            collision++;
                            linearProbing(hk, i);
                        }
                }
                else
                {
                        hk = hk % n;
                        linearProbing(hk, i);
                }
        }

        void output()
        {
                cout << "No. of Collision: " << collision << endl;
                cout << "Value " << "Index" << endl;
                for (int i = 0; i < n; i++)
                {
                        if (arrM[i] != 0)
                        {
                            cout << arrM[i] << " | " << i << endl;
                        }
                }
                cout << endl;
        }
};

int main()
{
    FoldBoundary fb;
    fb.input();
    if (!err)
    {
        fb.placing();
```

```
                fb.output();
        }
        return 0;
}
```



```
Enter how many elements you want to store: 5
Enter the data for 0 Index: 12345
Enter the data for 1 Index: 7878
Enter the data for 2 Index: 89831
Enter the data for 3 Index: 956192
Enter the data for 4 Index: 9819740
Array: 12345 7878 89831 956192 9819740
Enter Number of locations: 48
No. of Collision: 0
Value Index
956192 | 1
9819740 | 24
7878 | 26
12345 | 39
89831 | 43
```

Observation: In this practical session, I learned about different hashing techniques such as modulo division, mid-square, digit extraction, fold shift, and fold boundary.

Modulo Division**:** A method that computes the hash value by taking the remainder of the key divided by the size of the hash table to get the hash value and stores the element at that hash value index. If the calculated index is occupied, linear probing is used to find the next available index.

Mid-Square**:** This technique squares the key and takes the middle portion of the resulting digits as the hash value and stores the element at that hash value index. If the calculated index is occupied, linear probing is used to find the next available index.

Digit Extraction**:** This method involves extracting specific digits from the key based on user-defined positions. The extracted digits are then combined to form a hash value, then the hash value is divided by the size of the hash table to get the hash value and stores the element at that hash value index. If the calculated index is occupied, linear probing is used to find the next available index.

Fold Shift**:** This method involves breaking the key into smaller parts based on the length of the hash table. The smaller parts are summed together, and the final hash value is obtained by ignoring the remainder if any and by taking the remainder of this sum divided by the size of the table to get the hash value and stores the element at that hash value index. If the calculated index is occupied, linear probing is used to find the next available index.

Fold Boundary**:** Similar to fold shift, this method also breaks the key into parts but ensures that parts are padded with zeros if they are shorter than a defined length. It reverses the first and last key parts and sums them and ignore the remainder if any to get the hash value, which is then used to store the element at that hash value index. If the calculated index is occupied, linear probing is used to find the next available index.

Linear Probing:

Linear Probing is collision resolution technique were if a collision or a conflict is occurred then the result of the hash value + 1 index memory location is checked, if it is empty then add the element at that location and if not then again add + 1 and check memory location. If the location goes out of bound then do hash value % the size of the table and hash value is empty or not, if empty then add the element else add + 1 and check memory location.