**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:**

**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)

    {

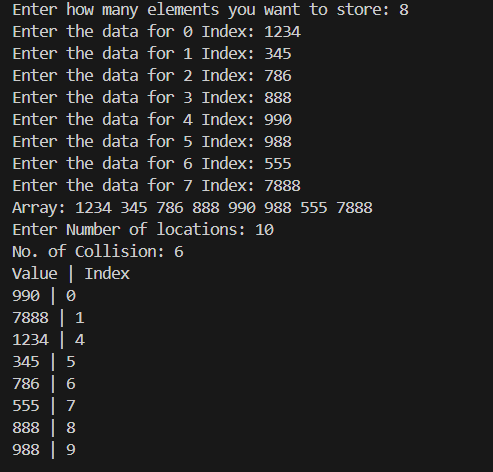
        md.placing();

        md.output();

    }

    return 0;

}



1. 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)

{

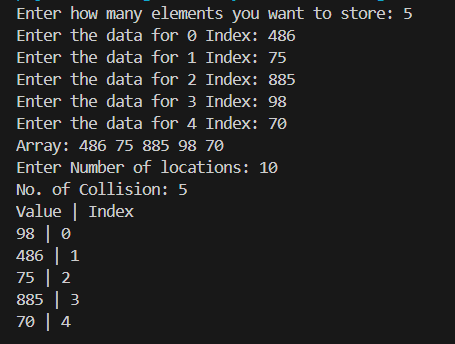
ms.placing();

ms.output();

}

return 0;

}



1. 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)

{

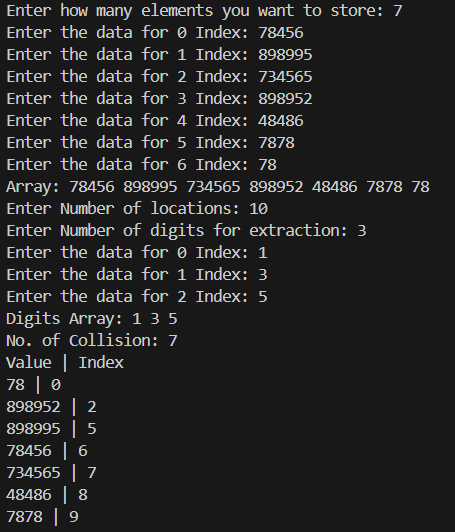
de.placing();

de.output();

}

return 0;

}



1. 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)

{

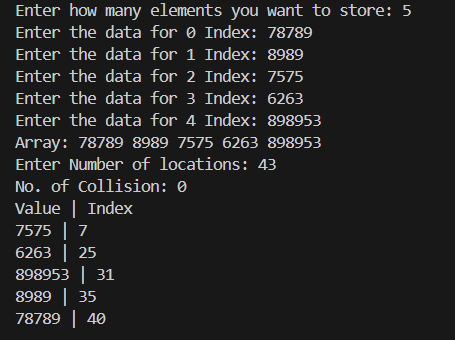
fs.placing();

fs.output();

}

return 0;

}



1. 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)

{

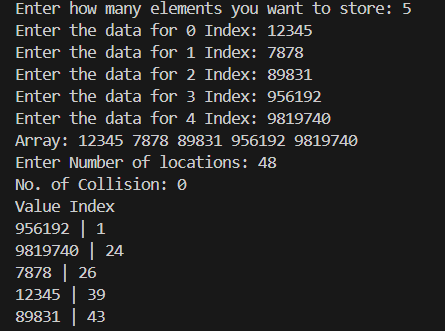
fb.placing();

fb.output();

}

return 0;

}



**Observation**: In this practical session I learned about different sorting techniques such as bubble sort, insertion sort, selection sort, quick sort, radix sort, merge sort.

Bubble Sort: Simple but inefficient for large datasets; repeatedly swaps adjacent elements. Best for small or simple tasks.

Insertion Sort: Builds a sorted array one element at a time; efficient for small or nearly sorted datasets. Slower for large unsorted arrays.

Selection Sort: Finds the minimum element and places it at the beginning; easy to understand but inefficient for larger arrays. Best for small datasets.

Quick Sort: It works by selecting a "pivot" element and partitioning the array into two halves: one with elements less than the pivot and one with elements greater than it. This process is repeated recursively to sort the given array, making it particularly effective for large datasets.

Radix Sort: Sorts numbers digit by digit; non-comparative and efficient for integers. It is not suitable for -ve nos.

Merge Sort: Divides and conquers by merging sorted subarrays; it is much more faster but requires extra space. Great for large datasets.