#pragma once

#include <iostream>

#include <time.h>

#include <vector>

using namespace std;

class HashEntry

{

public:

int key;

int value;

HashEntry()

{

key = -1;

value = -1;

}

HashEntry(int x, int y)

{

key = x;

value = y;

}

};

class HashMap

{

private:

int size;

HashEntry\*\* table;

int\* random\_arr;

public:

HashMap()

{

srand(time(0));

size = 125;

table = new HashEntry \* [size];

for (int i = 0; i < size; ++i)

table[i] = nullptr;

random\_arr = new int[size];

for (int i = 0; i < size; ++i)

random\_arr[i] = 0;

seedRandomArr();

}

HashMap(int size2)

{

srand(time(0));

size = size2;

table = new HashEntry \* [size];

for (int i = 0; i < size; ++i)

table[i] = nullptr;

random\_arr = new int[size];

for (int i = 0; i < size; ++i)

random\_arr[i] = 0;

seedRandomArr();

}

void seedRandomArr()

{

int random1;

for (int i = 0; i < size; i++)

{

random1 = rand() % size;

random\_arr[i] = random1;

}

}

int hashFunction(int key)

{

int hash = key % size;

return hash;

}

void Insert(int key, int value)

{

int h\_index = hashFunction(key);

bool found = 0;

while (table[h\_index] != nullptr && !found)

{

if (table[h\_index]->key == key)

found = true;

else

h\_index = (h\_index + 1) % size;

}

if (found)

{

std::cerr << "Duplicate items are not allowed." << std::endl;

return;

}

HashEntry\* ptr = new HashEntry();

ptr->key = key;

ptr->value = value;

table[h\_index] = ptr;

}

int Search(int key)

{

int h\_index = hashFunction(key);

while (table[h\_index] != nullptr)

{

if (table[h\_index]->key == key)

return table[h\_index]->value;

else

h\_index = (h\_index + 1) % size;

}

return -1;

}

void Remove(int key)

{

int h\_index = hashFunction(key);

while (table[h\_index] != nullptr)

{

if (table[h\_index]->key == key)

{

delete table[h\_index];

table[h\_index] = nullptr;

return;

}

h\_index = (h\_index + 1) % size;

}

std::cerr << "Key not found. Cannot remove." << std::endl;

}

~HashMap()

{

for (int i = 0; i < size; ++i)

{

if (table[i] != nullptr)

delete table[i];

}

delete[] table;

}

void Insert2(int key, int value)

{

int h\_index = hashFunction(key);

int probeIndex = 0;

while (table[h\_index] != nullptr && probeIndex < size - 1 && table[h\_index]->key != key)

{

h\_index = (h\_index + random\_arr[probeIndex]) % size;

probeIndex++;

}

if (table[h\_index] != NULL && table[h\_index]->key == key)

{

table[h\_index]->value = value;

}

else

{

table[h\_index] = new HashEntry(key, value);

}

}

};

class HashNode

{

public:

int key;

HashNode\* next;

HashNode(const int& k) : key(k), next(nullptr) {}

};

class CustomHashTable

{

public:

vector<HashNode\*> table;

int tableSize;

CustomHashTable(int size) : tableSize(size)

{

table.resize(size, nullptr);

}

int hashFunction(int key)

{

int hash = key % tableSize;

return hash;

}

void insert(const int& key)

{

int index = hashFunction(key);

HashNode\* newNode = new HashNode(key);

if (!table[index])

{

table[index] = newNode;

}

else

{

HashNode\* entry = table[index];

while (entry->next && entry->key != key)

{

entry = entry->next;

}

if (entry->key == key)

{

delete newNode;

}

else

{

entry->next = newNode;

}

}

}

//bool search(int key)

//{

// int index = hashFunction(key);

// HashNode\* entry = table[index];

// while (entry)

// {

// if (entry->key == key)

// {

// return true;

// }

// entry = entry->next;

// }

// return false;

//}

void search()

{

display();

}

void remove(const int& key)

{

int index = hashFunction(key);

HashNode\* entry = table[index];

HashNode\* prev = nullptr;

while (entry)

{

if (entry->key == key)

{

if (prev)

{

prev->next = entry->next;

}

else

{

table[index] = entry->next;

}

delete entry;

return;

}

prev = entry;

entry = entry->next;

}

}

void display() const

{

for (int i = 0; i < tableSize; ++i)

{

cout << "Bucket " << i << ": ";

HashNode\* entry = table[i];

if (!entry)

{

cout << "Empty" << endl;

}

else

{

while (entry)

{

cout << entry->key;

if (entry->next)

{

cout << " -> ";

}

entry = entry->next;

}

cout << endl;

}

}

}

~CustomHashTable()

{

for (int i = 0; i < tableSize; i++)

{

HashNode\* entry = table[i];

while (entry)

{

HashNode\* prev = entry;

entry = entry->next;

delete prev;

}

}

}

};

struct Student

{

string CNIC;

int roll;

};

int main() {

//Linear Probing

cout << "Hash Table with Linear Probing\n";

HashMap map(10);

map.Insert(1, 100);

map.Insert(2, 200);

map.Insert(3, 300);

map.Insert(12, 1200);

cout << "Value for key 1: " << map.Search(1) << endl;

cout << "Value for key 2: " << map.Search(2) << endl;

cout << "Value for key 3: " << map.Search(3) << endl;

cout << "Value for key 12: " << map.Search(12) << endl;

cout << "Value for key 4: " << map.Search(4) << endl;

map.Remove(2);

cout << "Value for key 2 after removal: " << map.Search(2) << endl;

map.Insert(1, 110);

cout << "\n\nHash Table with Chaining\n";

CustomHashTable map2(10);

cout << "Inserting values 1-5\n";

map2.insert(1);

map2.insert(2);

map2.insert(3);

map2.insert(4);

map2.insert(5);

cout << "\nDisplaying values after searching\n";

map2.search();

cout << "\nRemoving value 2 and inserting value 6-15\n\n";

map2.remove(2);

map2.insert(6);

map2.insert(7);

map2.insert(8);

map2.insert(9);

map2.insert(10);

map2.insert(11);

map2.insert(12);

map2.insert(13);

map2.insert(14);

map2.insert(15);

cout << "Final Display\n";

map2.search();

cout << "\n\nTask 2\n";

Student students[450];

CustomHashTable studentHash(50);

for (int i = 0; i <450; i++)

{

students[i].roll = i + 17551;

students[i].CNIC = "123-24234-3423" + i;

studentHash.insert(students[i].roll);

}

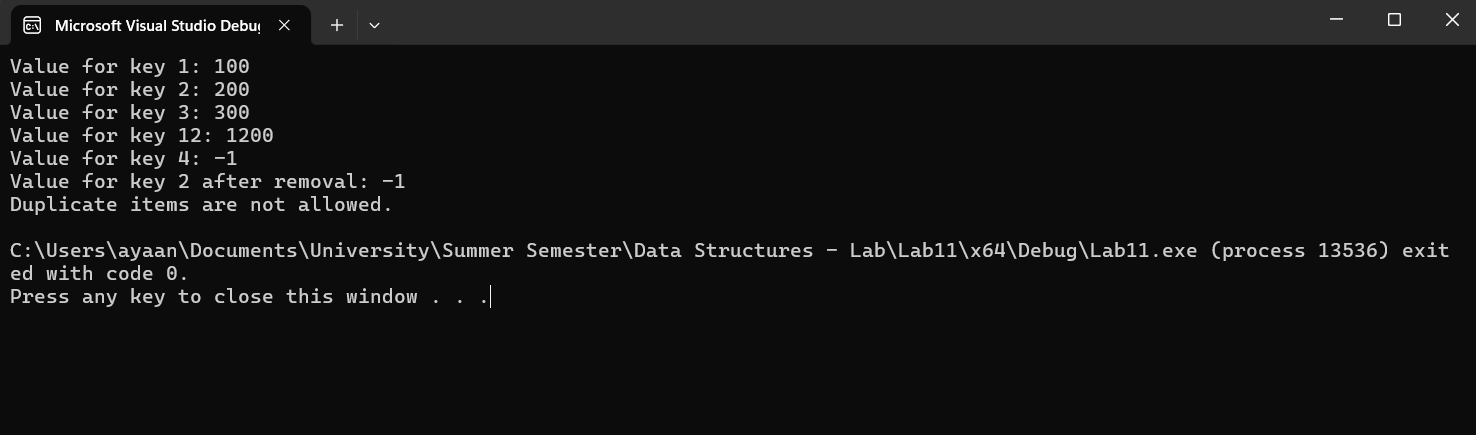
studentHash.display();

return 0;

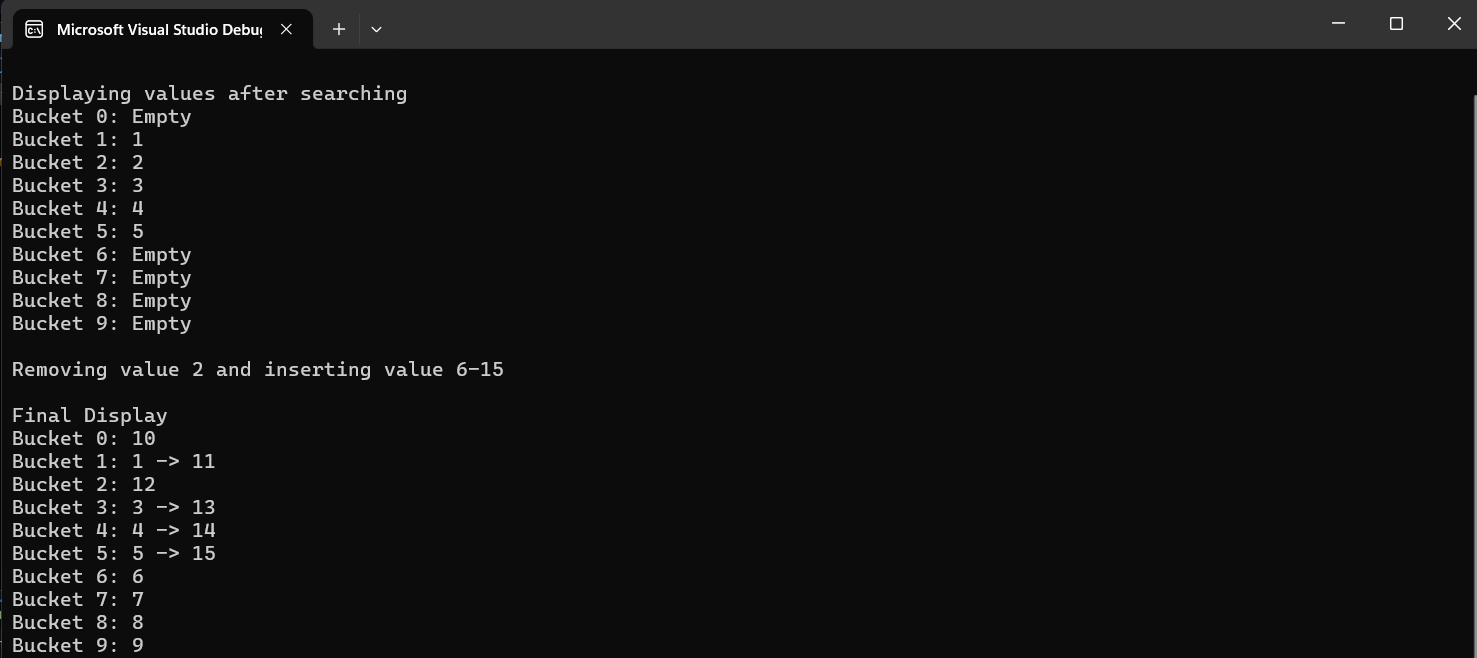
}

Task 01:

Linear Probing



Chaining



Task 02:

