

Алгоритми та структури даних

**Лабораторна робота №6**

**“Дослідження структури даних хеш-таблиця”**

Робота : Лещинського Б.Д. группа КА-07

17.03.2021

Варіант 13

**Мета роботи:**

Ознайомитись і дослідити структуру даних хеш-таблиця. Набути навичок реалізації хеш-таблиці за методом ланцюжків мовою програмування C++, познайомитись з використанням STL контейнерів на прикладі unordered\_map та порівняти власну реалізацію з готовим бібліотечним рішенням.

**Хід виконання роботи:**

**Завдання:**

Написати програму для реалізації структури даних хеш-таблиця, яка дозволить проводити дуже швидкий пошук даних у наборі структур свого варіанту.

1. Реалізувати функцію для генерації випадкового великого числа, що буде унікальним ключем-ідентифікатором даних у хеш-таблиці.

2. Реалізувати хеш-таблицю за методом ланцюжків:

2.1 Створити структуру Data для зберігання різнотипних даних відповідно до варіанту, вона буде використовуватись у якості значень для хеш-таблиці. Реалізувати генерацію випадкових даних для полів структури у пустому конструкторі Data().

2.2 Створити структуру HashNode для зберігання ключа та значення

2.3 Створити структуру LinkedList з підтримкою додавання елементів

2.4 Створити структуру HashTable, яка буде містити масив зв’язних списків bucketsArray статичного розміру M = 10000 та реалізувати в ній основні функції:

• hash(key) знаходження хеш-функції від ключа

• insert(key, value) додавання значення під відповідним ключем (заміна у випадку існування ключа)

• find(key) знаходження значення під відповідним ключем, функція повертає вказівник на знайдений елемент або NULL

• erase(key) видалення значення під відповідним ключем

• size() знаходження кількості елементів в хеш-таблиці Всі функції повинні працювати за O(1).

3. Оптимізувати хеш-таблицю, додавши динамічну зміну розміру масиву bucketsArray в залежності від loadFactor (максимально можлива завантаженість таблиці):

3.1 Початковий розмір масиву bucketsArray встановити невеликим, наприклад m = 8, при перевищенні значення loadFactor динамічно збільшувати розмір (наприклад в 2 рази)

3.2 Початкове значення loadFactor тимчасово обрати рівним 8.5 4. Провести тестування, використавши вказану нижче функцію testHashTable(). Обрати оптимальне значення loadFactor. Перевірити правильність та швидкість роботи, порівнявши з готовим бібліотечним рішенням STL unordered\_map.

Необов’язкові завдання для отримання додаткових балів:

• Реалізувати ще одну хеш-таблицю, де в якості ключів використати тип string

• Реалізувати ще одну хеш-таблицю за методом відкритої адресації

**Код програми(стандартне завдання) :**

#include <iostream>

#include <cstdlib>

#include <string>

#include <ctime>

#include <unordered\_map>

using namespace std;

long long generateRandLong() {

long long generated = rand()%9 + 1;

for (int i = 1; i < 10; i++) {

generated \*= 10;

generated += rand() % 10;

}

return generated;

}

string possibleNames[] = { "KPI", "Sheva", "Mogila", "UKU" };

struct Data {

string uniName;

int numberStudents;

bool tyrnicety;

Data() {

uniName = possibleNames[rand() % 4];

numberStudents = rand()%10001;

tyrnicety = rand() % 2;

}

};

struct HashNode {

Data uni;

long long key;

HashNode\* next;

HashNode\* previous;

};

struct LinkedList {

int entitiesList = 0;

HashNode\* first = nullptr;

HashNode\* last = nullptr;

void push\_back(long long key, Data uni) {

HashNode\* newLast = new HashNode;

newLast->uni = uni;

newLast->key = key;

if (entitiesList == 0) {

newLast->next = nullptr;

newLast->previous = nullptr;

first = newLast;

last = newLast;

}

else{

last->next = newLast;

newLast->previous = last;

newLast->next = nullptr;

last = newLast;

}

entitiesList++;

}

Data\* get\_byKey(long long key) {

HashNode\* searchNode = first;

if (searchNode == nullptr) return NULL;

for (int i = 0; i < entitiesList; i++) {

if (searchNode->key == key) {

return &searchNode->uni;

}

else {

searchNode = searchNode->next;

}

}

return NULL;

}

bool delete\_byKey(long long key) {

HashNode\* popingNode = first;

if (popingNode == nullptr) return NULL;

for (int i = 0; i < entitiesList; i++) {

if (popingNode->key == key) {

if (entitiesList == 1) {

delete popingNode;

first = nullptr;

last = nullptr;

entitiesList--;

return true;

}

else if (popingNode->previous == nullptr) {

popingNode->next->previous = nullptr;

first = popingNode->next;

delete popingNode;

entitiesList--;

return true;

}

else if (popingNode->next == nullptr) {

popingNode->previous->next = nullptr;

last = popingNode->previous;

delete popingNode;

entitiesList--;

return true;

}

else {

popingNode->previous->next = popingNode->next;

popingNode->next->previous = popingNode->previous;

delete popingNode;

entitiesList--;

return true;

}

}

else {

popingNode = popingNode->next;

}

}

return false;

}

HashNode\* pop\_front() {

if (entitiesList == 0) {

return NULL;

}

else if (entitiesList == 1) {

HashNode\* popedNode = first;

first = nullptr;

last = nullptr;

entitiesList--;

return popedNode;

}

else {

HashNode\* popedNode = first;

popedNode->next->previous = nullptr;

first = popedNode->next;

entitiesList--;

return popedNode;

}

}

};

struct HashTable {

double loadFactor = 0.8;

long M = 1024;

LinkedList\* bucketsArray = new LinkedList[M];

int entities = 0;

long long p = 9149658775000477;

int a = rand() % 1000;

int b = rand() % 1000;

int hash(long long key) {

return ((a \* key + b) % p) % M;

}

void insert(long long key, Data value) {

if (size() >= (double)M \* loadFactor) {

reshapeTable();

}

int hashInd = hash(key);

if (bucketsArray[hashInd].delete\_byKey(key)) {

bucketsArray[hashInd].push\_back(key, value);

}

else {

bucketsArray[hashInd].push\_back(key, value);

entities++;

}

}

Data\* find(long long key) {

int findHashInd = hash(key);

return bucketsArray[findHashInd].get\_byKey(key);

}

void erase(long long key) {

int eraseHashInd = hash(key);

if(bucketsArray[eraseHashInd].delete\_byKey(key)){

entities--;

}

}

int size() {

return entities;

}

void reshapeTable() {

M \*= 2;

entities = 0;

LinkedList\* oldBucketsArray = bucketsArray;

bucketsArray = new LinkedList[M];

for (int i = 0; i < M / 2; i++)

{

HashNode\* currentNode = oldBucketsArray[i].first;

if (currentNode == nullptr) continue;

do {

insert(currentNode->key, currentNode->uni);

if (currentNode->next != nullptr)

currentNode = currentNode->next;

else break;

} while (true);

}

delete[] oldBucketsArray;

}

};

bool testHashTable()

{

const int iters = 500000;

const int keysAmount = iters \* 1;

// generate random keys:

long long\* keys = new long long[keysAmount];

long long\* keysToInsert = new long long[iters];

long long\* keysToErase = new long long[iters];

long long\* keysToFind = new long long[iters];

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

{

keys[i] = generateRandLong();

}

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

{

keysToInsert[i] = keys[generateRandLong() % keysAmount];

keysToErase[i] = keys[generateRandLong() % keysAmount];

keysToFind[i] = keys[generateRandLong() % keysAmount];

}

// test my HashTable:

HashTable hashTable;

clock\_t myStart = clock();

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

{

hashTable.insert(keysToInsert[i], Data());

}

int myInsertSize = hashTable.size();

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

{

hashTable.erase(keysToErase[i]);

}

int myEraseSize = hashTable.size();

int myFoundAmount = 0;

for (int i = 0; i < iters; i++) {

if (hashTable.find(keysToFind[i]) != NULL)

{

myFoundAmount++;

}

}

clock\_t myEnd = clock();

float myTime = (float(myEnd - myStart)) / CLOCKS\_PER\_SEC;

// test STL hash table:

unordered\_map<long long, Data> unorderedMap;

clock\_t stlStart = clock();

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

{

unorderedMap.insert({ keysToInsert[i], Data() });

}

int stlInsertSize = unorderedMap.size();

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

{

unorderedMap.erase(keysToErase[i]);

}

int stlEraseSize = unorderedMap.size();

int stlFoundAmount = 0;

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

{

if (unorderedMap.find(keysToFind[i]) != unorderedMap.end())

{

stlFoundAmount++;

}

}

clock\_t stlEnd = clock();

float stlTime = (float(stlEnd - stlStart)) / CLOCKS\_PER\_SEC;

cout << "My HashTable:" << endl;

cout << "Time: " << myTime << ", size: " << myInsertSize << " - " << myEraseSize <<

", found amount: " << myFoundAmount << endl;

cout << "STL unordered\_map:" << endl;

cout << "Time: " << stlTime << ", size: " << stlInsertSize << " - " << stlEraseSize

<< ", found amount: " << stlFoundAmount << endl << endl;

delete keys;

delete keysToInsert;

delete keysToErase;

delete keysToFind;

if (myInsertSize == stlInsertSize && myEraseSize == stlEraseSize && myFoundAmount ==

stlFoundAmount)

{

cout << "The lab is completed" << endl;

return true;

}

cerr << ":(" << endl;

return false;

}

int main(){

srand(time(NULL));

testHashTable();

return 0;

}

**Код програми(тип значення ключа - стрінг) :**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include <cstdlib>

#include <string>

#include <ctime>

#include <unordered\_map>

#include <cmath>

using namespace std;

long long generateRandLong() {

long long generated = rand() % 9 + 1;

for (int i = 1; i < 10; i++) {

generated \*= 10;

generated += rand() % 10;

}

return generated;

}

string generate\_string\_key() {

string randStr = "";

for (int i = 0; i < 10; i++)

randStr = randStr + char(rand() % 23 + 97);

return randStr;

}

string possibleNames[] = { "KPI", "Sheva", "Mogila", "UKU" };

struct Data {

string uniName;

int numberStudents;

bool tyrnicety;

Data() {

uniName = possibleNames[rand() % 4];

numberStudents = rand() % 10001;

tyrnicety = rand() % 2;

}

};

struct HashNode {

Data uni;

string key;

HashNode\* next;

HashNode\* previous;

};

struct LinkedList {

int entitiesList = 0;

HashNode\* first = nullptr;

HashNode\* last = nullptr;

void push\_back(string& key, Data uni) {

HashNode\* newLast = new HashNode;

(newLast->uni).numberStudents = uni.numberStudents;

(newLast->uni).tyrnicety = uni.tyrnicety;

(newLast->uni).uniName = uni.uniName;

newLast->key = key;

if (entitiesList == 0) {

newLast->next = nullptr;

newLast->previous = nullptr;

first = newLast;

last = newLast;

}

else {

last->next = newLast;

newLast->previous = last;

newLast->next = nullptr;

last = newLast;

}

entitiesList++;

}

Data\* get\_byKey(string& key) {

HashNode\* searchNode = first;

if (searchNode == nullptr) return NULL;

for (int i = 0; i < entitiesList; i++) {

if (searchNode->key == key) {

return &searchNode->uni;

}

else {

searchNode = searchNode->next;

}

}

return NULL;

}

bool delete\_byKey(string& key) {

HashNode\* popingNode = first;

if (popingNode == nullptr) return NULL;

for (int i = 0; i < entitiesList; i++) {

if (popingNode->key == key) {

if (entitiesList == 1) {

delete popingNode;

first = nullptr;

last = nullptr;

entitiesList--;

return true;

}

else if (popingNode->previous == nullptr) {

popingNode->next->previous = nullptr;

first = popingNode->next;

delete popingNode;

entitiesList--;

return true;

}

else if (popingNode->next == nullptr) {

popingNode->previous->next = nullptr;

last = popingNode->previous;

delete popingNode;

entitiesList--;

return true;

}

else {

popingNode->previous->next = popingNode->next;

popingNode->next->previous = popingNode->previous;

delete popingNode;

entitiesList--;

return true;

}

}

else {

popingNode = popingNode->next;

}

}

return false;

}

HashNode\* pop\_front() {

if (entitiesList == 0) {

return NULL;

}

else if (entitiesList == 1) {

HashNode\* popedNode = first;

first = nullptr;

last = nullptr;

entitiesList--;

return popedNode;

}

else {

HashNode\* popedNode = first;

popedNode->next->previous = nullptr;

first = popedNode->next;

entitiesList--;

return popedNode;

}

}

};

struct HashTable {

double loadFactor = 0.8;

long M = 1000;

LinkedList\* bucketsArray = new LinkedList[M];

int entities = 0;

int a = rand() % 52 + 1;

int b = rand() % 52 + 1;

long long hash(string& key) {

long len = key.length();

int newK = key[0] \* key[len / 4] \* key[len - 1] \* a;

return newK % M;

}

void insert(string& key, Data value) {

if (size() >= M \* loadFactor) {

reshapeTable();

}

long long hashInd = hash(key);

if (bucketsArray[hashInd].delete\_byKey(key)) {

bucketsArray[hashInd].push\_back(key, value);

}

else {

bucketsArray[hashInd].push\_back(key, value);

entities++;

}

}

Data\* find(string& key) {

long long findHashInd = hash(key);

return bucketsArray[findHashInd].get\_byKey(key);

}

void erase(string& key) {

long long eraseHashInd = hash(key);

if (bucketsArray[eraseHashInd].delete\_byKey(key)) {

entities--;

}

}

int size() {

return entities;

}

void reshapeTable() {

M \*= 2;

LinkedList\* reshapedBuckets = new LinkedList[M];

HashNode\* transNode = new HashNode;

for (int i = 0; i < M / 2; i++) {

while (bucketsArray[i].entitiesList != 0) {

transNode = bucketsArray[i].pop\_front();

long long new\_hashInd = hash(transNode->key);

reshapedBuckets[new\_hashInd].push\_back(transNode->key, transNode->uni);

}

}

delete[] bucketsArray;

bucketsArray = reshapedBuckets;

}

};

bool testHashTable()

{

const int iters = 500000;

const int keysAmount = iters \* 1;

string\* keys = new string[keysAmount];

string\* keysToInsert = new string[iters];

string\* keysToErase = new string[iters];

string\* keysToFind = new string[iters];

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

{

keys[i] = generate\_string\_key();

}

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

{

keysToInsert[i] = keys[generateRandLong() % keysAmount];

keysToErase[i] = keys[generateRandLong() % keysAmount];

keysToFind[i] = keys[generateRandLong() % keysAmount];

}

HashTable hashTable;

clock\_t myStart = clock();

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

{

hashTable.insert(keysToInsert[i], Data());

}

int myInsertSize = hashTable.size();

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

{

hashTable.erase(keysToErase[i]);

}

int myEraseSize = hashTable.size();

int myFoundAmount = 0;

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

{

if (hashTable.find(keysToFind[i]) != NULL)

{

myFoundAmount++;

}

}

clock\_t myEnd = clock();

float myTime = (float(myEnd - myStart)) / CLOCKS\_PER\_SEC;

unordered\_map<std::string, Data> unorderedMap;

clock\_t stlStart = clock();

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

{

unorderedMap.insert({ keysToInsert[i], Data() });

}

int stlInsertSize = unorderedMap.size();

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

{

unorderedMap.erase(keysToErase[i]);

}

int stlEraseSize = unorderedMap.size();

int stlFoundAmount = 0;

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

{

if (unorderedMap.find(keysToFind[i]) != unorderedMap.end())

{

stlFoundAmount++;

}

}

clock\_t stlEnd = clock();

float stlTime = (float(stlEnd - stlStart)) / CLOCKS\_PER\_SEC;

cout << "My HashTable:" << endl;

cout << "Time: " << myTime << ", size: " << myInsertSize << " -" << myEraseSize << ", found amount: " << myFoundAmount << endl;

cout << "STL unordered\_map:" << endl;

cout << "Time: " << stlTime << ", size: " << stlInsertSize << " -" << stlEraseSize << ", found amount: " << stlFoundAmount << endl

<< endl;

delete[] keys;

delete[] keysToInsert;

delete[] keysToErase;

delete[] keysToFind;

if (myInsertSize == stlInsertSize && myEraseSize == stlEraseSize && myFoundAmount == stlFoundAmount)

{

cout << "The lab is completed" << endl;

return true;

}

cerr << ":(" << endl;

return false;

}

int main(){

srand(time(NULL));

testHashTable();

return 0;

**Код програми(таблиця з відкритою адресацією) :**

#include <iostream>

#include <cstdlib>

#include <string>

#include <ctime>

#include <unordered\_map>

using namespace std;

long long generateRandLong() {

long long generated = rand() % 9 + 1;

for (int i = 1; i < 10; i++) {

generated \*= 10;

generated += rand() % 10;

}

return generated;

}

string possibleNames[] = { "KPI", "Sheva", "Mogila", "UKU" };

struct Data {

string uniName;

int numberStudents;

bool tyrnicety;

Data() {

uniName = possibleNames[rand() % 4];

numberStudents = rand() % 10001;

tyrnicety = rand() % 2;

}

};

int greatest\_common\_divider(long long a, long long b) {

if (a == 1 || b == 1 || a==0 || b==0 ) {

return 1;

}

if (a % b == 0)

return b;

if (b % a == 0)

return a;

if (a > b)

return greatest\_common\_divider(a % b, b);

return greatest\_common\_divider(a, b % a);

}

struct HashTable {

struct DataContainer {

long long key;

Data\* uni = nullptr;

bool isDeleted = false;

};

long M = 2;

long entities = 0;

DataContainer\* bucketsArray = new DataContainer[M];

long long p1 = 9149658775000477;

int a1 = rand() % 1000;

int b1 = rand() % 1000;

long long hash1(long long key) {

return ((a1 \* key + b1) % p1) % M;

}

long long p2 = 7775194452319117;

int a2 = rand() % 1000;

int b2 = rand() % 1000;

long long hash2(long long key) {

long long hashVal1 = hash1(key);

long long hashVal2 = ((a2 \* key + b2) % p2) %M;

if (greatest\_common\_divider(hashVal1, hashVal2) != 1) {

hashVal2 /= greatest\_common\_divider(hashVal1, hashVal2);

}

return hashVal2;

}

void insert(long long key, Data univer) {

if (entities >= M) {

reshapeTable();

}

int hashInd = hash1(key);

if (find(key) != NULL) return;

else if (bucketsArray[hashInd].isDeleted == false && bucketsArray[hashInd].uni == nullptr) {

bucketsArray[hashInd].uni = new Data;

\*(bucketsArray[hashInd].uni) = univer;

bucketsArray[hashInd].key = key;

entities++;

}

else {

long long step = hash2(key);

int i = 0;

do {

i++;

hashInd = (hashInd + i \* step) % M;

} while (bucketsArray[hashInd].uni != nullptr || bucketsArray[hashInd].isDeleted == true);

bucketsArray[hashInd].uni = new Data;

\*(bucketsArray[hashInd].uni) = univer;

bucketsArray[hashInd].key = key;

entities++;

}

}

Data\* find(long long key) {

int hashInd = hash1(key);

if (bucketsArray[hashInd].key == key && bucketsArray[hashInd].isDeleted == true) return NULL;

else if (bucketsArray[hashInd].key == key && bucketsArray[hashInd].isDeleted == false) {

return bucketsArray[hashInd].uni;

}

else {

long long startedInd = hashInd;

long long step = hash2(key);

hashInd = (hashInd + step) % M;

int i = 1;

do {

if (bucketsArray[hashInd].key == key && bucketsArray[hashInd].isDeleted == false) return bucketsArray[hashInd].uni;

else if(bucketsArray[hashInd].key == key && bucketsArray[hashInd].isDeleted == true) return NULL;

i++;

hashInd = (hashInd + i\*step) % M;

} while (bucketsArray[hashInd].uni != nullptr && hashInd != startedInd);

return NULL;

}

}

void erase(long long key) {

if (find(key) == NULL) return;

else {

int hashInd = hash1(key);

if (bucketsArray[hashInd].key == key && bucketsArray[hashInd].isDeleted == false) {

bucketsArray[hashInd].isDeleted = true;

entities--;

}

else {

long long step = hash2(key);

hashInd = (hashInd + step) % M;

int i = 1;

do {

if (bucketsArray[hashInd].key == key && bucketsArray[hashInd].isDeleted == false) {

bucketsArray[hashInd].isDeleted = true;

entities--;

}

i++;

hashInd = (hashInd + i \* step) % M;

} while (bucketsArray[hashInd].uni != nullptr);

}

}

}

long size() {

return entities;

}

void reshapeTable(){

M \*= 2;

entities = 0;

DataContainer\* oldBucketsArray = bucketsArray;

bucketsArray = new DataContainer[M];

for (int i = 0; i < M / 2; i++) {

DataContainer currentContainer = oldBucketsArray[i];

if (currentContainer.isDeleted == true) continue;

else {

insert(currentContainer.key, \*currentContainer.uni); /// \*current uni

}

}

delete[] oldBucketsArray;

}

};

bool testHashTable()

{

const int iters = 500000;

const int keysAmount = iters \* 1;

// generate random keys:

long long\* keys = new long long[keysAmount];

long long\* keysToInsert = new long long[iters];

long long\* keysToErase = new long long[iters];

long long\* keysToFind = new long long[iters];

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

{

keys[i] = generateRandLong();

}

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

{

keysToInsert[i] = keys[generateRandLong() % keysAmount];

keysToErase[i] = keys[generateRandLong() % keysAmount];

keysToFind[i] = keys[generateRandLong() % keysAmount];

}

// test my HashTable:

HashTable hashTable;

clock\_t myStart = clock();

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

{

hashTable.insert(keysToInsert[i], Data());

}

int myInsertSize = hashTable.size();

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

{

hashTable.erase(keysToErase[i]);

}

int myEraseSize = hashTable.size();

int myFoundAmount = 0;

for (int i = 0; i < iters; i++) {

if (hashTable.find(keysToFind[i]) != NULL)

{

myFoundAmount++;

}

}

clock\_t myEnd = clock();

float myTime = (float(myEnd - myStart)) / CLOCKS\_PER\_SEC;

// test STL hash table:

unordered\_map<long long, Data> unorderedMap;

clock\_t stlStart = clock();

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

{

unorderedMap.insert({ keysToInsert[i], Data() });

}

int stlInsertSize = unorderedMap.size();

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

{

unorderedMap.erase(keysToErase[i]);

}

int stlEraseSize = unorderedMap.size();

int stlFoundAmount = 0;

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

{

if (unorderedMap.find(keysToFind[i]) != unorderedMap.end())

{

stlFoundAmount++;

}

}

clock\_t stlEnd = clock();

float stlTime = (float(stlEnd - stlStart)) / CLOCKS\_PER\_SEC;

cout << "My HashTable:" << endl;

cout << "Time: " << myTime << ", size: " << myInsertSize << " - " << myEraseSize <<

", found amount: " << myFoundAmount << endl;

cout << "STL unordered\_map:" << endl;

cout << "Time: " << stlTime << ", size: " << stlInsertSize << " - " << stlEraseSize

<< ", found amount: " << stlFoundAmount << endl << endl;

delete keys;

delete keysToInsert;

delete keysToErase;

delete keysToFind;

if (myInsertSize == stlInsertSize && myEraseSize == stlEraseSize && myFoundAmount ==

stlFoundAmount)

{

cout << "The lab is completed" << endl;

return true;

}

cerr << ":(" << endl;

return false;

}

int main()

{

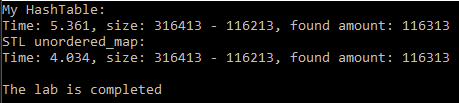
srand(time(NULL));

testHashTable();

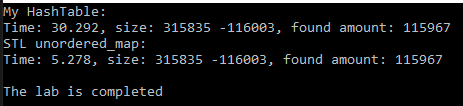
return 0;

}

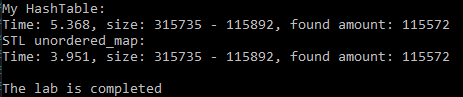
**Результати роботи програми(стандартне завдання):**



**Результати роботи програми(тип значення ключа - стрінг):**



**Результати роботи програми(таблиця з відкритою адресацією):**



Висновки:

Я ознайомився і дослідиі структуру даних хеш-таблиця. Набув навичок реалізації хеш-таблиці за методом ланцюжків та методом відкритої адресації з використанням подвійної буферицації(більш детально про реалізацію саме цього метода зможу розказати на захисті лабораторної) мовою програмування C++, познайомитись з використанням STL контейнерів на прикладі unordered\_map та порівняти власну реалізацію з готовим бібліотечним рішенням.