**Лабораторна робота №1**

**З дисципліни «Алгоритми та структури даних»**

***«Дослідження структур даних зв’язний список та динамічний масив»***

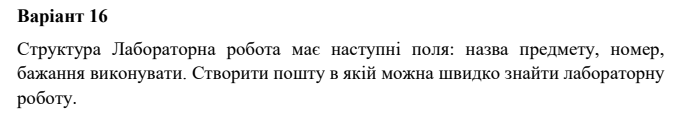
*Виконав  
студент I курсу  
групи ДА-12  
Ніколаєв Роман*

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.

*Варіант 16*

**Хід виконання завдання:**



#include <iostream>

#include <cmath>

#include <ctime>

#include <unordered\_map>

using namespace std;

#define P 9149658775000477

#define A 24

#define B 11

struct Data

{

char name;

int num;

bool motivation;

Data(){

name = (char)(rand() % 26 + 65);

num = rand() % 10001;

motivation = rand() % 2;

};

};

struct HashNode

{

long long key;

Data value;

HashNode\* next;

};

struct LinkedList

{

HashNode\* head;

LinkedList();

void pushFront(Data, long long);

};

LinkedList::LinkedList()

{

head = nullptr;

}

void LinkedList::pushFront(Data newValue, long long newKey)

{

HashNode\* newNode = new HashNode();

newNode->value = newValue;

newNode->key = newKey;

newNode->next = head;

head = newNode;

}

struct HashTable

{

int realSize, capacity;

const int alpha = 2;

float loadFactor;

LinkedList\* bucketsArray;

HashTable();

int hash(long long);

void insert(long long, Data);

HashNode\* find(long long);

void erase(long long);

int size();

};

HashTable::HashTable()

{

capacity = 8;

realSize = 0;

loadFactor = 0.5;

bucketsArray = new LinkedList[capacity];

}

int HashTable::hash(long long keyToHash)

{

return ((A \* keyToHash + B) % P) % capacity;

}

void HashTable::insert(long long keyToInsert, Data valueToInsert)

{

if ((float)realSize / capacity > loadFactor) {

int oldCapacity = capacity;

capacity \*= alpha;

LinkedList\* oldBucketsArray = bucketsArray;

bucketsArray = new LinkedList[capacity];

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

{

if (oldBucketsArray[i].head != nullptr)

{

HashNode\* Node = oldBucketsArray[i].head;

while (Node->next != nullptr) {

int pos1 = hash(Node->key);

bucketsArray[pos1].pushFront(Node->value, Node->key);

Node = Node->next;

}

int pos1 = hash(Node->key);

bucketsArray[pos1].pushFront(Node->value, Node->key);

}

}

delete[] oldBucketsArray;

}

int pos = hash(keyToInsert);

HashNode\* Node = bucketsArray[pos].head;

if (Node != nullptr) {

while ((Node->next != nullptr) && (keyToInsert != Node->key))

Node = Node->next;

if (Node->key == keyToInsert)

Node->value = valueToInsert;

else {

bucketsArray[pos].pushFront(valueToInsert, keyToInsert);

realSize++;

}

}

else {

bucketsArray[pos].pushFront(valueToInsert, keyToInsert);

realSize++;

}

}

HashNode\* HashTable::find(long long keyToFind)

{

int pos = hash(keyToFind);

HashNode\* Node = bucketsArray[pos].head;

if (Node != nullptr) {

while ((Node->next != nullptr) && (keyToFind != Node->key))

Node = Node->next;

if (Node->key == keyToFind)

return Node;

else

return nullptr;

}

else

return nullptr;

}

void HashTable::erase(long long keyToErase)

{

int pos = hash(keyToErase);

HashNode\* Node = bucketsArray[pos].head;

if (Node != nullptr) {

if (Node->key == keyToErase) {

bucketsArray[pos].head = Node->next;

realSize--;

delete Node;

}

else if (Node->next != nullptr) {

while ((Node->next != nullptr) && (keyToErase != Node->next->key))

Node = Node->next;

if ((Node->next != nullptr) && (Node->next->key == keyToErase)) {

HashNode\* save = Node->next;

Node->next = Node->next->next;

realSize--;

delete save;

}

}

}

}

int HashTable::size()

{

return realSize;

}

long long generateRandLong()

{

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

for (int i = 1; i < 13; i++)

key = key \* 10 + rand() % 10;

return key;

}

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]) != nullptr)

{

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()

{

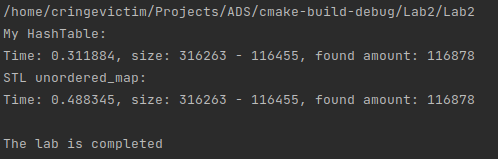
srand(time(nullptr));

testHashTable();

return 0;

}

**Результат виконання:**



**Висновки:**

*Виконуючи дану роботу було набуто навичок реалізації хеш-таблиці*

**Контрольні питання:**

1. *Хеш-таблиця це структура для зберігання даних, котра дає миттєвий доступ до елементів за їх унікальним ключем. Звичайний масив, на відміну від хеш-таблиці, не є нескінченним, хоч і має миттєвий доступ до елементів. Список, на відміну від хеш-таблиці, має серйозне обмеження в швидкості пошуку елементів, через необхідність перевіряти кожний послідовний елемент, доки необхідний не буде знайдений.*
2. *Хеш-функція повинна максимально уникати генерацію однакових індексів, під якими зберігаються дані в структурі, щоб уникати колізій у структурі, які сильно сповільнюють операції. Універсальне хешування – вид хешування, при якому хеш-функція випадково вибирається із заданого сімейства.*
3. *Колізія це по суті поверння хеш-функцією однакового значення при різних вхідних даних. Для вирішення колізії можна використовувати декілька методів. Один з них – метод ланцюжків, за якого хеш-таблиці представлена у вигляді динамічоного масиву зв’язних списків. За цього методу асимптотика залежить від завантаженості таблиці, через лінійність пошуку елементів у зв’язних списках, а також вказівники на ноди зв’язних списків займають додадкову пам’ять. Ще один метод – метод відкритої адресації: ітерація по масиву відбувається швидше, але кластеризація сповільнює операціїї з таблицею.*
4. *Значення loadFactor є відношенням к-ті елементів до максимального розміру таблиці (*(float)realSize / capacity *в коді) і визначає поточний рівень заповненості таблиці. Значеня maxLoadFactor (*loadFactor *в коді) визначає максимальну степінь заповненості таблиці, після досягнення якого відбувається розширення таблиці. Якщо значення занадто мале – часті релокації сповільнять роботу таблиці, а якщо занадто велике – перезавантаженість таблиці спричинить часті колізії, що також сповільнить її роботу.*
5. *Для різних типів даних використовується сімейство універсальних хеш-функцій. Для отримання хешу від строки, модна використати функцію* H = (s0+s1k2+...+snkn)mod(p)*, де sn – символи строки, n – розмір строки, k – константа, більша за кількість можливих символів в строці, p – велике просте число. Алгоритм Рабіна-Карпа – алгоритм пощука стрічки, який шукає шаблон (підстроку) в тексті за допомогою хешування.*