

Звіт

Лабораторна робота №2

“Дослідження структури даних

хеш-таблиця”

Студента группи ДA-12

Краковича Павла Дмитровича

Київ – 2022

1. Мета роботи

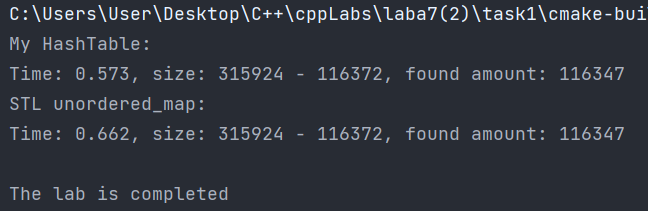
Ознайомитись і дослідити структуру даних хеш-таблиця. Набути навичок реалізації хеш-таблиці за методом ланцюжків мовою програмування C++, познайомитись з використанням STL контейнерів на прикладі unordered\_map та порівняти власну реалізацію з готовим бібліотечним рішенням.

2. Варіант роботи - №9

3. Хід виконання роботи

№1

* Структура Викладач має наступні поля: ім’я, предмет, якість мікрофону. Створити кафедру на якій можна швидко знайти викладача.
* Результат роботи програми:



* Лістинг(код) програми:
  + main.cpp

//#include "HashTable\_Closed\_Addressing.h"  
#include "HashTable\_Linear\_Probing.h"  
#include "ctime"  
#include "cmath"  
#include <unordered\_map>  
  
using namespace std;  
  
  
unsigned long long generateRandLong()  
{  
 unsigned long long result = 0;  
 int iters = rand() % 3 + 12;  
 for (int i = 0; i < iters ; i++) {  
 result += (rand() % 10) \* pow(10, i);  
 }  
 result += ((rand() % 9 + 1) \* pow(10, iters));  
 return result;  
}  
  
bool testHashTable()  
{  
 const int iters = 500000;  
 const int keysAmount = iters \* 1;  
  
 // generate random keys:  
 auto\* keys = new unsigned long long[keysAmount];  
  
 auto\* keysToInsert = new unsigned long long[iters];  
 auto\* keysToErase = new unsigned long long[iters];  
 auto\* keysToFind = new unsigned 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:  
  
  
 // ---------------------------------------------------------------------  
 // if user wants to use default HashTable, then he should use next code:  
  
// HashTable\_Closed\_Addressing hashTable;  
  
 // else  
  
 HashTable\_Linear\_Probing 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])) {  
 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()});  
 }  
 size\_t stlInsertSize = unorderedMap.size();  
 for (int i = 0; i < iters; i++)  
 {  
 unorderedMap.erase(keysToErase[i]);  
 }  
 size\_t 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;  
}

* + HashList.h

//  
// Created by User on 25.04.2022.  
//  
  
#ifndef MAIN\_CPP\_HASHLIST\_H  
#define MAIN\_CPP\_HASHLIST\_H  
  
#include <iostream>  
using namespace std;  
  
struct Data {  
 string name;  
 string subject;  
 string microphoneQuality;  
};  
  
struct HashNode {  
 long long key;  
 Data data;  
 HashNode\* next;  
};  
  
struct LinkedList {  
 HashNode\* head;  
 int size;  
  
 LinkedList();  
  
 void push\_front(long long int Key, Data newData);  
  
 bool remove(unsigned long long int Key);  
  
 Data\* get(unsigned long long int Key);  
  
 ~LinkedList();  
};  
  
#endif //MAIN\_CPP\_HASHLIST\_H

* + HashList.cpp

//  
// Created by User on 25.04.2022.  
//  
#include "HashList.h"  
  
LinkedList::LinkedList() {  
 head = nullptr;  
 size = 0;  
}  
  
void LinkedList::push\_front(long long int Key, Data newData) {  
 HashNode\* node = new HashNode();  
 node->data = newData;  
 node->key = Key;  
 node->next = head;  
 head = node;  
 size++;  
}  
  
bool LinkedList::remove(unsigned long long int Key) {  
 HashNode\* node = head;  
 HashNode\* temp = nullptr;  
 while (node) {  
 if (node->key == Key) {  
 if (temp != nullptr) {  
 temp->next = node->next;  
 }  
 else {  
 head = node->next;  
 }  
 delete node;  
 size--;  
 return true;  
 }  
 temp = node;  
 node = node->next;  
 }  
 return false;  
}  
  
Data\* LinkedList::get(unsigned long long int Key) {  
 HashNode\* node = head;  
 while (node) {  
 if (node->key == Key) {  
 return &node->data;  
 }  
 node = node->next;  
 }  
 return nullptr;  
}  
LinkedList::~LinkedList() {  
 HashNode\* node = head;  
 while (head) {  
 node = head;  
 head = head->next;  
 delete node;  
 }  
}

* + HashTable\_Separate\_Chaining.h

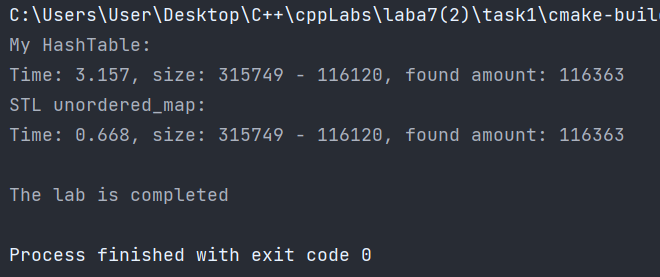
//  
// Created by User on 06.06.2022.  
//  
#include "HashList.h"  
  
#ifndef MAIN\_CPP\_HASHTABLE\_H  
#define MAIN\_CPP\_HASHTABLE\_H  
  
struct HashTable\_Separate\_Chaining {  
 LinkedList\* Buckets;  
 int capacity;  
 int Size;  
 float LoadFactor;  
 float LoadCapacity;  
  
 HashTable\_Separate\_Chaining();  
  
 unsigned long long hash(unsigned long long key) const;  
  
 void reallocate();  
  
 void insert(long long key, const Data& data);  
  
 Data\* find(unsigned long long key) const;  
  
 void erase(unsigned long long key);  
  
 int size() const;  
  
 ~HashTable\_Separate\_Chaining();  
};  
  
#endif //MAIN\_CPP\_HASHTABLE\_H

* + HashTable\_Separate\_Chaining.cpp

//  
// Created by User on 06.06.2022.  
//  
  
#include "HashTable\_Separate\_Chaining.h"  
  
  
HashTable\_Separate\_Chaining::HashTable\_Separate\_Chaining() {  
 capacity = 10;  
 Size = 0;  
 LoadFactor = 0.0f;  
 LoadCapacity = 0.9f;  
 Buckets = new LinkedList[capacity];  
}  
  
unsigned long long HashTable\_Separate\_Chaining::hash(unsigned long long key) const  
{  
 return ((12 \* key - 2) % 376258005738947) % capacity;  
}  
  
void HashTable\_Separate\_Chaining::reallocate()  
{  
 int previousCapacity = capacity;  
 capacity \*= 2.0f;  
  
 LinkedList\* TempBuckets;  
 TempBuckets = new LinkedList[capacity];  
  
 for (size\_t i = 0; i < previousCapacity; i++)  
 {  
 LinkedList& list = Buckets[i];  
 HashNode\* node = list.head;  
 while (node)  
 {  
 size\_t index = hash(node->key);  
 LinkedList& tempList = TempBuckets[index];  
 if (list.get(node->key)) {  
 tempList.push\_front(node->key, node->data);  
 }  
 node = node->next;  
 }  
 }  
  
 delete[] Buckets;  
 Buckets = TempBuckets;  
}  
  
  
void HashTable\_Separate\_Chaining::insert(long long key, const Data& data)  
{  
 if ((LoadFactor = Size / capacity) > LoadCapacity) {  
 reallocate();  
 }  
  
 unsigned long long index = hash(key);  
 LinkedList& list = Buckets[index];  
 if (!list.get(key)) {  
 list.push\_front(key, data);  
 Size++;  
 }  
}  
  
Data\* HashTable\_Separate\_Chaining::find(unsigned long long key) const  
{  
 size\_t index = hash(key);  
 LinkedList& list = Buckets[index];  
 return list.get(key);  
}  
  
void HashTable\_Separate\_Chaining::erase(unsigned long long key)  
{  
 size\_t index = hash(key);  
 LinkedList& list = Buckets[index];  
 if (list.remove(key)) {  
 Size--;  
 }  
}  
  
int HashTable\_Separate\_Chaining::size() const {  
 return Size;  
}  
  
HashTable\_Separate\_Chaining::~HashTable\_Separate\_Chaining()  
{  
 delete[] Buckets;  
}

№2 (Додаткове завдання 1)

* Реалізувати ще одну хеш-таблицю за методом відкритої адресації:
* Результат роботи програми:



* Лістинг(код) програми:
  + HashTable\_Linear\_Probing.h

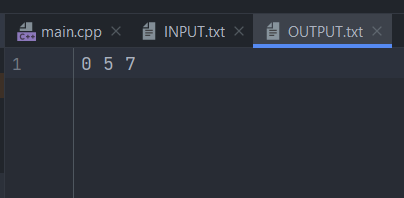
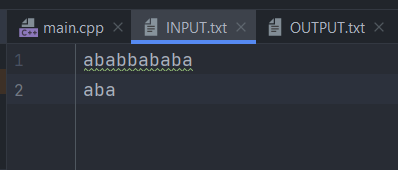
//  
// Created by User on 06.06.2022.  
//  
  
#ifndef MAIN\_CPP\_HASHTABLE\_LINEAR\_PROBING\_H  
#define MAIN\_CPP\_HASHTABLE\_LINEAR\_PROBING\_H  
#include <iostream>  
using namespace std;  
  
  
struct Data {  
 string name;  
 string subject;  
 string microphoneQuality;  
};  
  
struct HashNode {  
 long long key;  
 Data data;  
};  
  
struct Node  
{  
 int statusCode;  
 long long key;  
 Data data;  
};  
  
struct HashTable\_Linear\_Probing {  
 Node\* Buckets;  
 int capacity;  
 int Size;  
 float LoadFactor;  
 float LoadCapacity;  
  
 HashTable\_Linear\_Probing();  
  
 long long hash(long long key) const;  
  
 void reallocate();  
  
 void insert(long long key, const Data& data);  
  
 Data\* find(unsigned long long key) const;  
  
 void erase(unsigned long long key);  
  
 int size() const;  
  
 ~HashTable\_Linear\_Probing();  
};  
  
#endif //MAIN\_CPP\_HASHTABLE\_LINEAR\_PROBING\_H

* + HashTable\_Linear\_Probing.cpp

//  
// Created by User on 06.06.2022.  
//  
#include "HashTable\_Linear\_Probing.h"  
  
HashTable\_Linear\_Probing::HashTable\_Linear\_Probing() {  
 capacity = 10;  
 Size = 0;  
 LoadFactor = 0.0f;  
 LoadCapacity = 0.65f;  
 Buckets = new Node[capacity];  
 for (int i = 0; i < capacity; i++) {  
 Buckets[i].statusCode = 0;  
 }  
}  
  
long long HashTable\_Linear\_Probing::hash(long long key) const {  
 return ((12 \* key - 2) % 376258005738947) % capacity;  
}  
  
void HashTable\_Linear\_Probing::reallocate()  
{  
 int previousCapacity = capacity;  
 capacity \*= 2.0f;  
 Node\* TempBuckets = new Node[capacity];  
 for (size\_t i = 0; i < previousCapacity; i++)  
 {  
 Node& node = Buckets[i];  
 if (node.statusCode == 1)  
 {  
 size\_t index = hash(node.key);  
  
 while (TempBuckets[index].statusCode == 1) {  
 index = (index + 1) % capacity;  
 }  
  
 TempBuckets[index] = node;  
 }  
 }  
 Buckets = TempBuckets;  
}  
  
void HashTable\_Linear\_Probing::insert(long long key, const Data& data)  
{  
 if ((LoadFactor = Size / capacity) > LoadCapacity) {  
 reallocate();  
 }  
  
 size\_t index = hash(key);  
 while (Buckets[index].statusCode == 1)  
 {  
 Node& node = Buckets[index];  
 if (node.key == key)  
 {  
 node.data = data;  
 return;  
 }  
 index = (index + 1) % capacity;  
 }  
  
 Buckets[index] = {1, key, data};  
 Size++;  
}  
  
Data\* HashTable\_Linear\_Probing::find(unsigned long long key) const {  
 size\_t index = hash(key);  
 while (Buckets[index].statusCode != 0) {  
 Node& node = Buckets[index];  
 if (node.key == key) {  
 Data\* ptr = nullptr;  
 if (node.statusCode == 1)  
 ptr = &node.data;  
 return ptr;  
 }  
 index = (index + 1) % capacity;  
 }  
 return nullptr;  
}  
  
void HashTable\_Linear\_Probing::erase(unsigned long long key) {  
 size\_t index = hash(key);  
 while (Buckets[index].statusCode != 0) {  
 Node& node = Buckets[index];  
 if (node.key == key) {  
 if (node.statusCode == 1) {  
 node.statusCode = 2;  
 Size--;  
 }  
  
 }  
 index = (index + 1) % capacity;  
 }  
}  
  
int HashTable\_Linear\_Probing::size() const {  
 return Size;  
}  
  
HashTable\_Linear\_Probing::~HashTable\_Linear\_Probing() {  
 delete[] Buckets;  
}  
//

№3 (Додаткове завдання 2)

* Розв’язати задачу “Пошук зразка в тексті”:
* Результат роботи програми:



* Лістинг(код) програми:

#include <fstream>  
#include <cmath>  
using namespace std;  
  
void createAndFillFile(const string& filename, const string& S, const string& T)  
{  
 ofstream fin(filename);  
  
 fin << S;  
 fin << "\n";  
 fin << T;  
  
 fin.close();  
}  
  
void createEmptyFile(const string& filename)  
{  
 ofstream fout(filename);  
 fout.close();  
}  
  
void fillEmptyFile(const string& filename, const string& OUTPUT)  
{  
 ofstream fin(filename);  
  
 fin << OUTPUT;  
 fin << " ";  
  
 fin.close();  
}  
  
void search(const string& filename, string pattern, string str)  
{  
 string OUTPUT;  
 int strLen = str.length();  
 int patternLen = pattern.length();  
 int strHash = 0, patternLenHash = 0;  
 int x = 10/\*Константа, основа для степеня\*/,powerOfX = pow(x, patternLen - 1);  
 int i = 0, j = 0, k = 0; // Змiннi для циклiв  
  
 while (i < patternLen) { // Обчислюю значення хешу для строки та паттерну  
 strHash += str[i] \* pow(x, patternLen - i - 1);  
 patternLenHash += pattern[i] \* pow(x, patternLen - i - 1);  
 i++;  
 }  
 while (j < strLen - patternLen + 1) { // Головний цикл  
 if (strHash == patternLenHash) {  
 bool isHashEqual = true;  
 while (k < patternLen) {  
 if (pattern[k] != str[k + j]) {  
 isHashEqual = false;  
 break;  
 }  
 k++;  
 }  
 if (isHashEqual) {  
 OUTPUT = OUTPUT + to\_string(j) + " ";  
 }  
 }  
 strHash = (strHash - str[j] \* powerOfX) \* x + str[patternLen + j];  
 j++;  
 }  
 fillEmptyFile(filename,OUTPUT); // Передаю значення у функцiю та друкую у файлi  
}  
  
int main()  
{  
 string str = "ababbababa";  
 string T = "aba";  
  
 string InputFile = "../INPUT.txt";  
 string OutputFile = "../OUTPUT.txt";;  
 createAndFillFile(InputFile,str,T);  
 createEmptyFile(OutputFile);  
  
 search(OutputFile,T, str);  
 return 0;  
}

Висновки

Виконуючи лабораторну роботу №2 я ознайомвся, дослідив та реалізував структуру даних «Хеш-таблиця» на зв’язних списках та лінійному зондуванні. Познайомився з STL контейнерами на прикладі unordered\_map, навчився їх використовувати та порівняв зі своїмии реалізаціями.

Моя реалізація структури на списках виявилась доволі швидкою, та швидшою за STL реалізацію. Однак, реалізація на лінійному зондуванню виявилась набагато повільнішою через первинну кластеризацію, що можна було б виправити додавши подвійне хешування, або частково вирішити квадратичним зондуванням, але, я вирішив для наглядності проблеми кластеризації зробити саме лінійне зондування.

Виконуючи додаткове завдання я проаналізував алгоритм Рябіна-Карпа та написав свою реалізацію, яка працює набагато швидше за звичайне пошук по буквам. Для створення текстових файлів та роботи з ними, я використав бібліотеку <fstream>, про яку дізнався з курсу «Алгоритмів та програмування».

Вивчивши структуру даних «Хеш-таблиця» я зрозумів, що вона є незамінною для специфічних цілей. Також, я дізнався, що завдяки хеш функціям можливо перевіряти цілісніть файлів та шукати дані по хеш-сумі.

Контрольні питання

1) Навіщо потрібна структура даних хеш-таблиця? Чим вона відрізняється від звичайного масива, списка? Коли її варто застосовувати?

*Хеш-таблиця — це структура даних, яка зберігає дані в асоціативний спосіб. У хеш-таблиці дані зберігаються у форматі масиву, де кожне значення даних має своє унікальне значення індексу. Доступ до даних стає дуже швидким, якщо ми знаємо індекс потрібних даних.*

*Таким чином, він стає структурою даних, в якій операції вставки та пошуку виконуються дуже швидко, незалежно від розміру даних. Хеш-таблиця використовує масив як носій даних і використовує хеш-техніку для створення індексу, куди потрібно вставити або звідки розташувати елемент.*

2) Яким чином вибір хеш-функції впливає на швидкість роботи хеш-таблиці? Що таке універсальне хешування?

*Якщо обрати невдалу хеш-функцію, то під час роботи програми буде виникати багато колізій, що спричинить значне сповільнення швидкості роботи програми.*

*Універсальне хешування - це вид хешування, при якому використовується не одна конкретна хеш-функція, а відбувається вибір із заданого сімейства за випадковим алгоритмом. Такий підхід забезпечує рівномірне хешування: для чергового ключа ймовірності поміщення його в будь-яку комірку збігаються.*

3) Що таке колізії у хеш-таблицях? Які способи вирішення колізій існують, в чому переваги і недоліки кожного з них?

*Колізія хеш-функції – це випадок, коли два різних вхідних елементів таблиці мають однакове значення хешу. Колізії зустрічаються у різноманітних алгоритмах хешування, проте це є нормою й у «правильних» алгоритмах їх виникнення зведено до мінімального значення.*

*Для вирішення колізій використовують реалізації хеш-таблиці з використанням зв’язних списків, зондувань та подвійного хешування. Кожен з цих методів частково вирішує проблему колізій, але й містить свої недоліки. Так, додавання списків вирішує проблему з колізіями, але збільшує час роботи, бо потрібно проходити по списку та збільшує кількість пам’яті, потрібної для роботи таблиці.*

*Лінійне зондування також вирішує проблему колізій, але я дуже вимогливим до хеш-функції, бо колізії спричиняють появу кластерів з даними, що сповільнює процеси додавання, пошуку та видалення елементів. Квадратичне зондування – це модифікація лінійного зондування, але цей метод не вирішує повністю проблему кластеризації.*

*Подвійне хешування в поєднанні з іншими методами дуже ефективно вирішує проблему кластеризації, одак, є складним у реалізації.*

4) За що відповідає значення loadFactor? Як змінюється швидкість роботи хеш-таблиці від обраного значення maxLoadFactor?

*Значення loadFactor відповідає за поточний рівень завантаженості таблиці.*

*Значення MaxLoadFactor відповідає за максимально допустиме навантаження таблиці. Якщо встановлювати його завеликим, то можуть початися з’являтися проблеми з колізіями, а замале значення змусить таблицю розширюватися занадто часто, що теж сповільнює роботу програми.*

5) Як виконувати хешування різних типів даних? Як отримати хеш від строки? В чому суть алгоритма Рабіна-Карпа?

*Хешування від числових даних доволі просте. Ми використовуємо хеш-функцію для отримання значення хешу. Щоб, наприклад отримати хеш від рядкового типу даних, доцільно перетворити рядок символів у суму ASCII символів рядку. Так ми отримаємо числове значення рядка та зможемо перетворити його у хеш.*

*Алгоритм Рабіна — Карпа — це алгоритм пошуку рядка, який шукає шаблон, тобто підрядок у тексті, використовуючи хешування.*

*Алгоритм рідко використовується для пошуку одиночного шаблону, але має значну теоретичну важливість і дуже ефективний у пошуку збігів багатьох шаблонів однакової довжини. Для тексту довжини n і шаблону довжини m його середній і найкращий час виконання дорівнює O(n) при правильному виборі хеш-функції, але в гіршому випадку він має ефективність O(nm), що є однією з причин того, чому він не надто широко використовується.*

*Для додатків, у яких допустимі помилкові спрацьовування при пошуку, тобто, коли деякі зі знайдених входження шаблону насправді можуть не відповідати шаблону, алгоритм Рабіна — Карпа працює за гарантований час O(n) і при виборі рандомізованої хеш-функції.*

*Також алгоритм має унікальну особливість знаходити будь-який із заданих k рядків однакової довжини в середньому (при правильному виборі хеш-функції) за час O(n) незалежно від розміру k.*

*Одне з найпростіших практичних застосувань алгоритму Рабіна-Карпа полягає у визначенні плагіату. Алгоритм Рабина — Карпа може швидко знайти у статті, що перевіряється, приклади входження деяких пропозицій з вихідних матеріалів. Для усунення чутливості алгоритму до невеликих відмінностей можна ігнорувати деталі, такі як регістр або пунктуація, їх видалення. Оскільки кількість рядків, які ми шукаємо, дуже велика, звичайні алгоритми пошуку одиночних рядків стають неефективними.*