МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

«Харківський Політехнічний Інститут»

Кафедра «Стратегічне управління»

Звіт з лабораторної роботи №6

«Хеш таблиці»

з дисципліни

"Алгоритми та структури даних"

Варіант №1

Перевірив:

ст. викл. каф. стратегічного

управління Мошко Є.О.

Виконала:

Беспалова Є.А.

гр. КН-321г

Харків 2022

**Тема лабораторної робот**: Хеш-таблиці та робота з ними

**Мета:** Вивчити роботу алгоритмів: прямої адресації, хеш-таблиці і відкритої адресації. Реалізувати перелічені вище алгоритми і провести їх порівняльний аналіз.

**Порядок виконання роботи:**

1. Написати алгоритми основних операцій (додавання, пошук і видалення) в Псевдокод для прямої адресації (метод ланцюжка), хеш-таблиці і відкритої адресації.
2. Реалізувати основні операції.
3. Провести аналіз цих алгоритмів і зробити висновок по їх застосовності.

**Завдання**

1. Нехай розмір хеш-таблиці дорівнює m = 20, а хеш-функція має вигляд: h(k) = [m(kA mod 1)], где А = ( 5 - l)/2. В які позиції потраплять ключі (див. таблицю, стовпець 3)?
2. Як буде виглядати хеш-таблиця з ланцюжками після того, як в неї послідовно помістили елементи з ключами (див. таблицю, стовпець 2) (в зазначеному порядку)? Число позицій в таблиці дорівнює 9, хеш-функція має вигляд h(k) = k mod 9.
3. Виконайте додавання ключів (в зазначеному порядку, див. таблицю стовпець 2) в хеш-таблицю з відкритою адресацією розміру m = 11. Для обчислення послідовності проб використовується лінійний метод з h'(k) = k mod m. Виконайте те ж завдання, якщо використовується квадратичний метод з тієї ж h', c1 = 1, c2 = 3, а також для подвійного хешування з h1 = h' и h2(k) = 1 + (k mod (m - 1)) .

Використовувати готові реалізації структур даних (наприклад, STL) заборонено, але можна використати реалізацію рядків (наприклад, std::string у C++).

Таблиця 1 – Варіант завдання

|  |  |  |
| --- | --- | --- |
| № | Масив | Ключі |
| 1 | 10, 22, 31, 4, 15, 28, 17, 88, 59 | 61, 62, 63, 64, 65 |

ЗМІСТ

[**1. ПРЯМА АДРЕСАЦІЯ 26**](#_Toc21160)

[**1.1 Вивести таблицю з файлу 26**](#_Toc26216)

[**1.2 Створення текстового файлу для ключа 26**](#_Toc20987)

[**1.3 Створення хеш-функції 27**](#_Toc1857)

[**1.4 Створення хеш таблиці 27**](#_Toc21716)

[**1.5 Вивод в консоль 27**](#_Toc6013)

[**1.6 Додавання елемента 28**](#_Toc19615)

[**1.7 Видалення елементу 30**](#_Toc7178)

[**1.8 Пошук елемента 31**](#_Toc4935)

[**2 ХЕШ ТАБЛИЦЯ З ЛАНЦЮЖКАМИ 34**](#_Toc29307)

[**2.1 Вивод таблиці з файлу 34**](#_Toc289)

[**2.2 Створення хеш-функції 35**](#_Toc25218)

[**2.3 Створення хеш таблиці 35**](#_Toc20183)

[**2.4 Створення вузлу 35**](#_Toc676)

[**2.5 Додавання елементу 36**](#_Toc19201)

[**2.6 Видалення елементу 38**](#_Toc12582)

[**2.7 Знайти число 41**](#_Toc17642)

[**3 ВІДКРИТА АДРЕСАЦІЯ 44**](#_Toc14799)

[**3.1 Створення хеш-функції 44**](#_Toc2717)

[**3.2 Створення хеш таблиці 44**](#_Toc25641)

[**3.3 Лінійний метод 44**](#_Toc30061)

[**3.3.1 Додавання елементу 46**](#_Toc27546)

[**3.3.2 Видалення елементу 48**](#_Toc14567)

[**3.3.3 Пошук елементу 50**](#_Toc12758)

[**3.4 Квадратичний метод 52**](#_Toc9431)

[**3.4.1 Додавання елементу 53**](#_Toc20719)

[**3.4.2 Видалення елементу 55**](#_Toc13822)

[**3.4.3 Пошук елементу 57**](#_Toc28203)

[**3.5.1 Подвійне хешування 60**](#_Toc23831)

[**3.5.1 Додавання елементу 60**](#_Toc24114)

[**3.5.2 Видалення елементу 62**](#_Toc22766)

[**3.5.3 Пошук елементу 64**](#_Toc12048)

[**4 АСИМТОТИЧНИЙ АНАЛІЗ 66**](#_Toc5117)

[**ВИСНОВКИ 68**](#_Toc12066)

# ПРЯМА АДРЕСАЦІЯ

* 1. Вивести таблицю з файлу

Приклад коду:

void HashTable::outFile(string path)

{

ifstream fin;

fin.open(path);

int element = 0;

if (!fin.is\_open())

{

return;

}

clearTable();

while (!fin.eof())

{

fin >> element;

addElement(element);

}

}

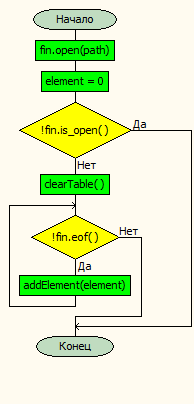


Рисунок 1.1.1 - Вивод таблиці з файлу

* 1. Створення текстового файлу для ключа

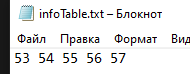


Рисунок 1.2.1 - Файл з ключем

1.3 Створення хеш-функції

Приклад коду:

int hashFunction(int key) {

return SIZE \* fmod(key \* A, 1);

}

1.4 Створення хеш таблиці

Приклад коду :

HashTable::HashTable()

{

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

{

table[i] = 0;

}

}

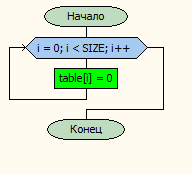


Рисунок 1.4.1 - заповнення хеш-таблиці

1.5 Вивод в консоль

Приклад коду:

void HashTable::printTable() const

{

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

{

if (table[i] != 0) {

cout << "Элемент " << i << ": " << table[i] << endl;

}

}

}

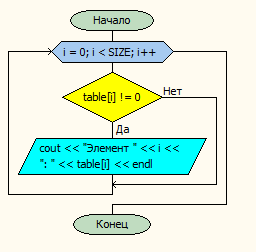


Рисунок 1.5.1 - блок-схема виводу в консоль

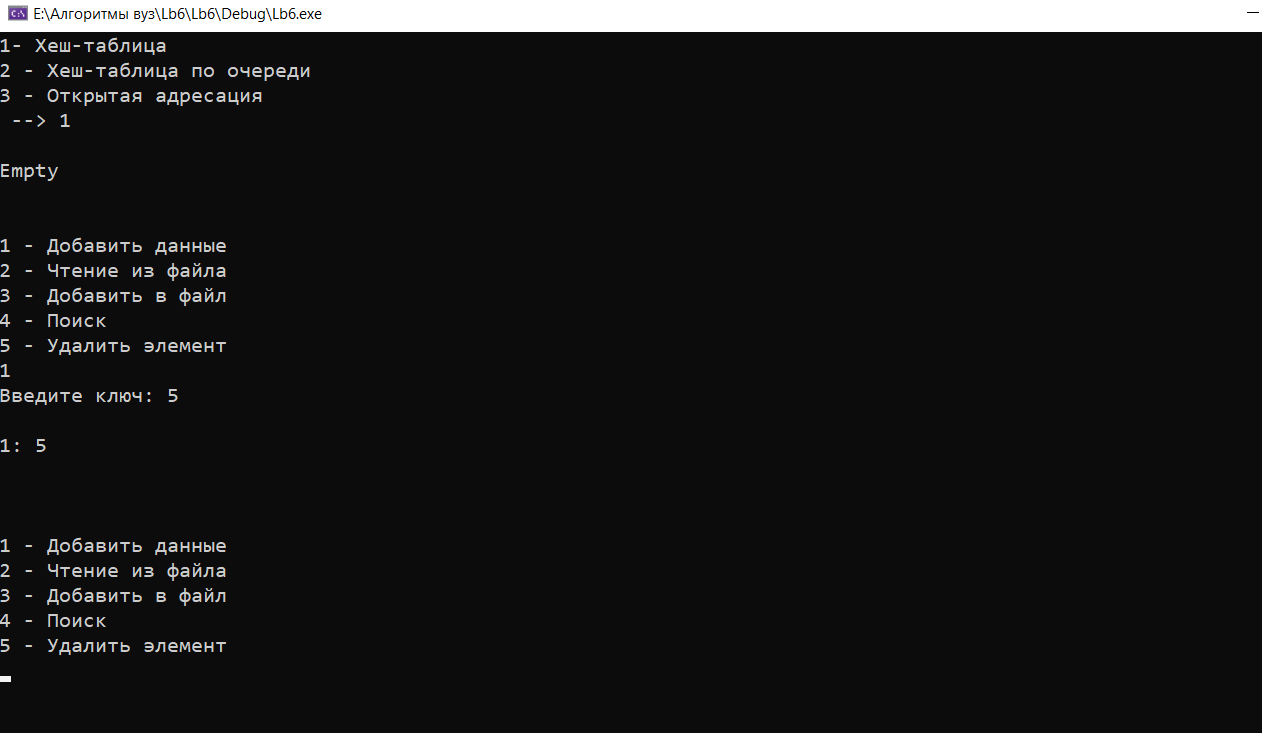


Рисунок 1.5.2 - вигдяд з консолі

1.6 Додавання елемента

Приклад коду:

void HashTable::addElement(int key)

{

table[hashFunction(key)] = key;

}

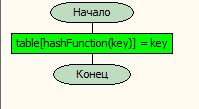


Рисунок 1.6.1 - додавання елементу

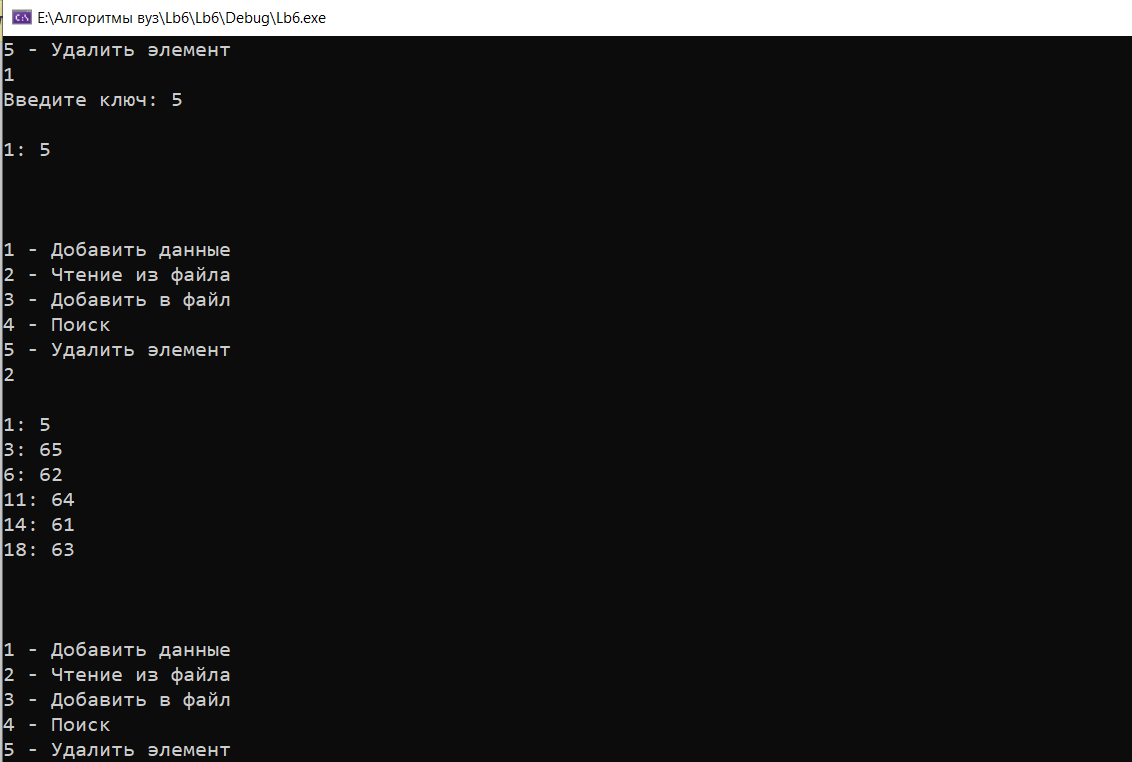


Рисунок 1.6.2 - вигляд з консолі

1.7 Видалення елементу

Приклад коду:

void HashTable::deleteElement(int key)

{

if (table[hashFunction(key)] == 0) {

cout << "Элемент не существует!" << endl;

}

else {

table[hashFunction(key)] = 0;

cout << "Элемент удалён!" << endl;

}

}

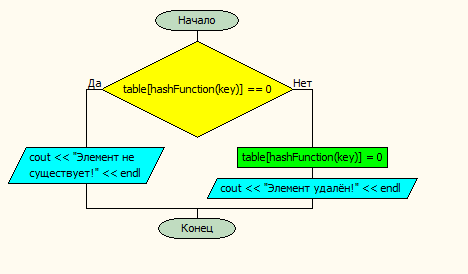


Рисунок 1.7.1 - видалення елементу

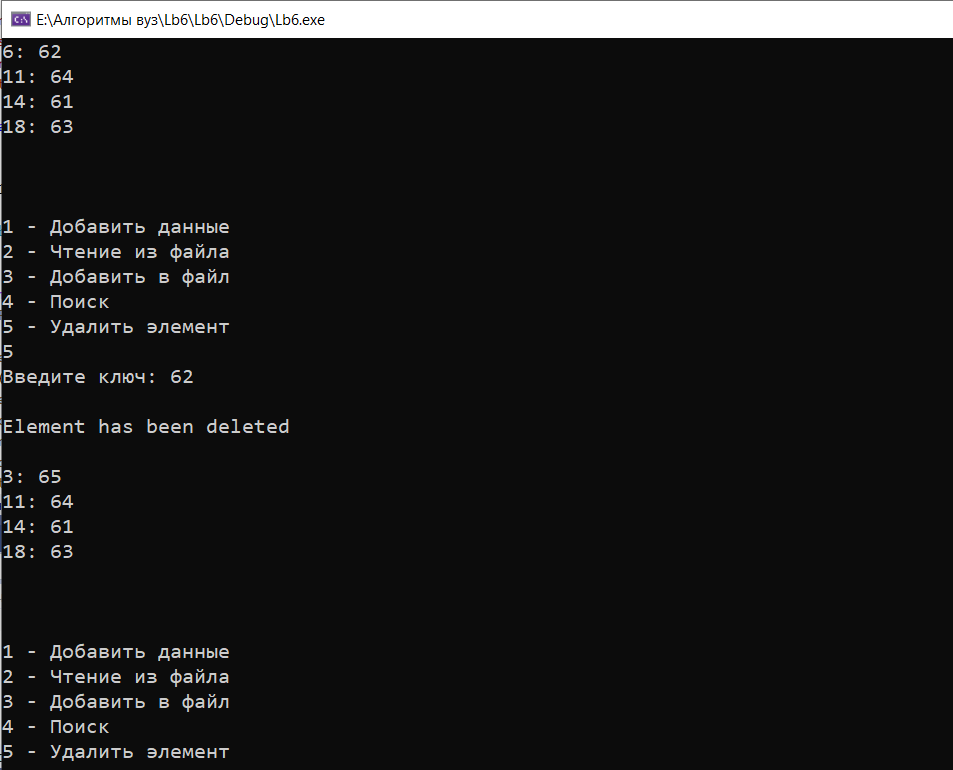


Рисунок 1.7.2 - виглад з консолі

1.8 Пошук елемента

Приклад коду:

int HashTable::searchElement(int key)

{

if (table[hashFunction(key)] == 0) {

return -1;

}

else if(table[hashFunction(key)] != key && table[hashFunction(key)] != 0) {

return -1;

}

else {

return hashFunction(key);

}

}

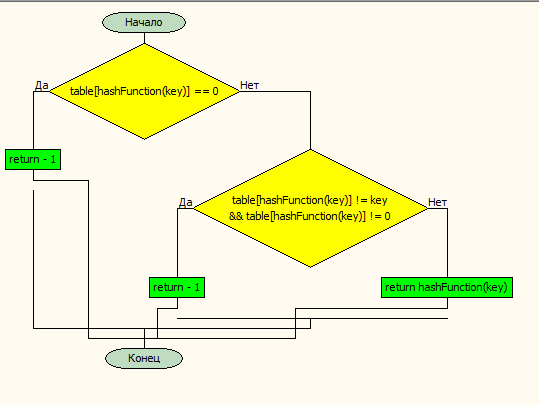


Рисунок 1.8.1 - Пошук елемента

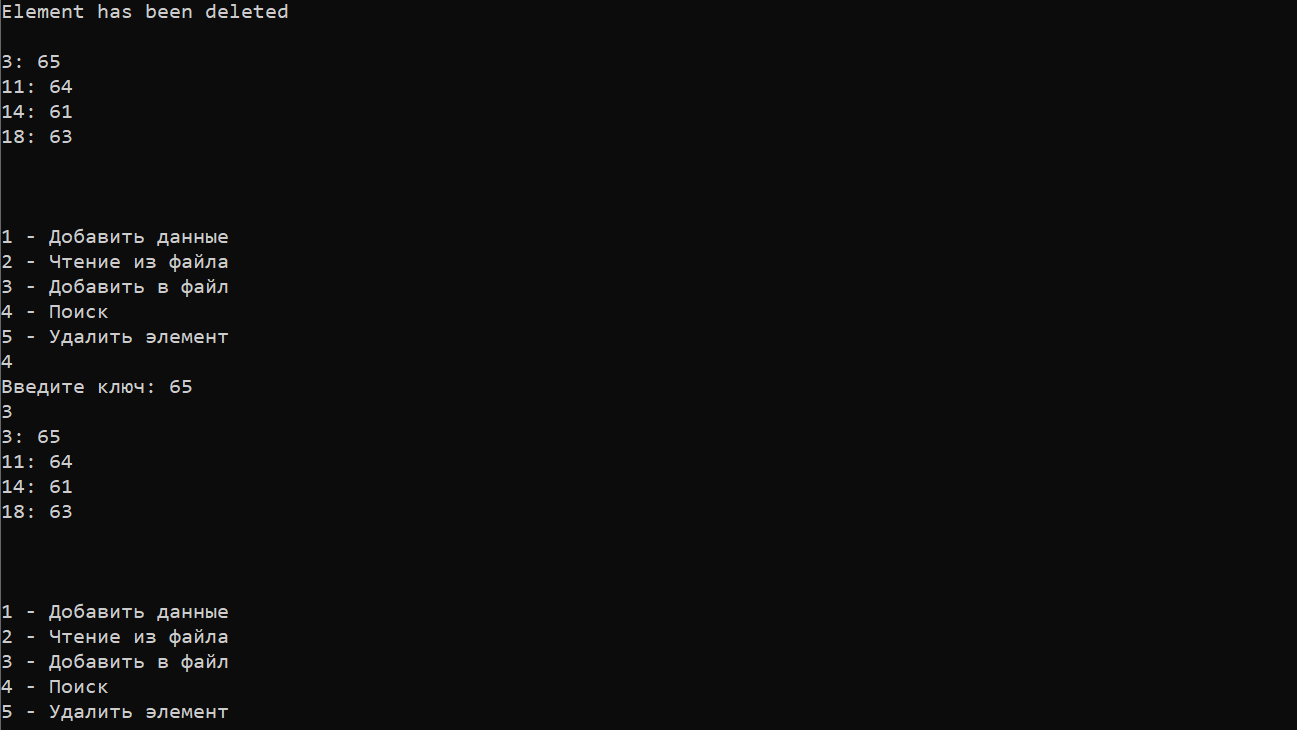


Рисунок 1.8.2 - вигляд з консолі

2 ХЕШ ТАБЛИЦЯ З ЛАНЦЮЖКАМИ

2.1 Вивод таблиці з файлу



Рисунок 2.1.1 - Ключи

Приклад коду для виводу із файлу:

void HTChain::outFile(string path)

{

ifstream fin;

fin.open(path);

if (!fin.is\_open()) {

return;

}

this->clearTable();

int numb = 0;

while (!fin.eof()) {

fin >> numb;

this->addElement(numb);

}

}

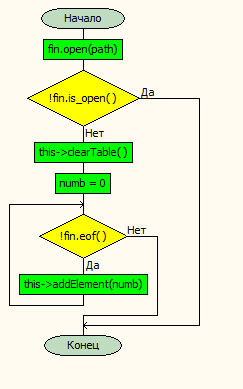


Рисунок 2.1.2 - блоксхема виводу із файлу

2.2 Створення хеш-функції

Приклад коду:

int hashFunction(int key) {

return fmod(key , SIZE);

}

2.3 Створення хеш таблиці

Приклад коду :

HashTable::HashTable()

{

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

{

table[i] = 0;

}

}

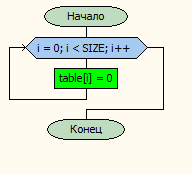


Рисунок 2.3.1 - заповнення хеш-таблиці

2.4 Створення вузлу

Приклад коду:

HashNode\* HTChain::createNode(int key)

{

HashNode\* newNode = new HashNode;

newNode->data = key;

newNode->next = NULL;

return newNode;

}

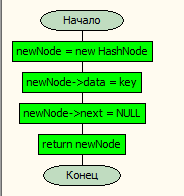


Рисунок 2.4.1 - Створення нового вузлу

2.5 Додавання елементу

Приклад коду:

void HTChain::addElement(int key)

{

int index = hashFunction(key);

if (hashTable[index] == NULL) {

hashTable[index] = createNode(key);

}

else {

HashNode\* temp = hashTable[index];

while (temp->next != NULL)

{

temp = temp->next;

}

temp->next = createNode(key);

}

}

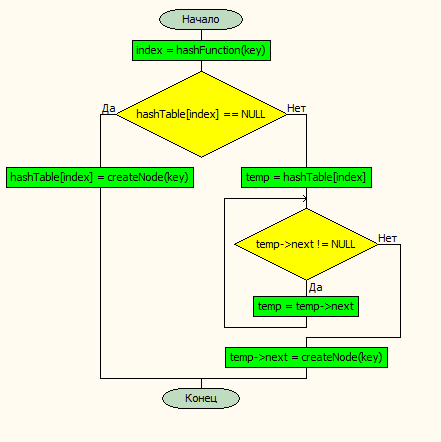


Рисунок 2.5.1 - додавання елементу

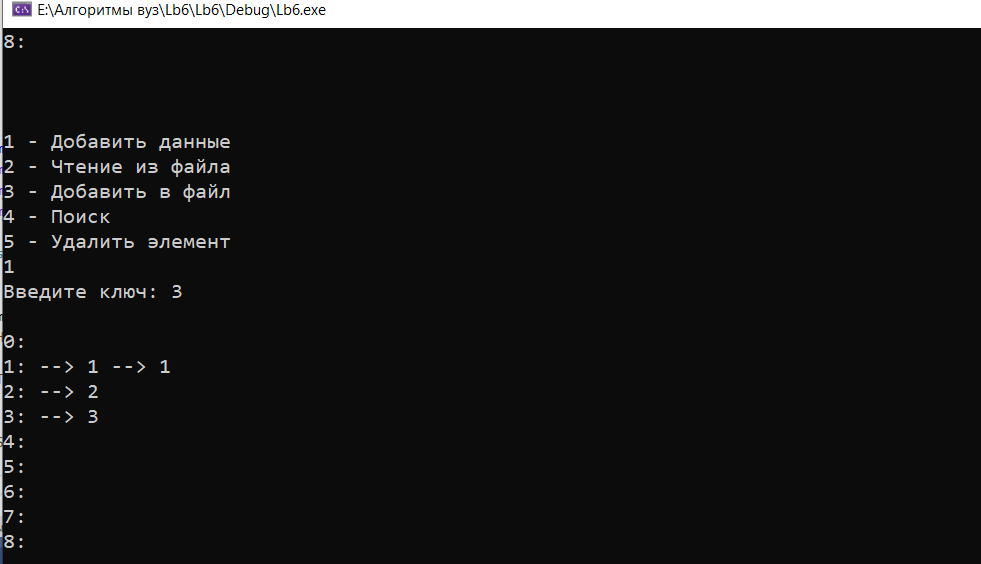


Рисунок 2.5.2 - вигляд з консолі

2.6 Видалення елементу

Приклад коду:

void HTChain::deleteElement(int key)

{

int hashCode = hashFunction(key);

int index = 0;

HashNode\* temp = hashTable[hashCode];

while (temp->data != key)

{

index++;

temp = temp->next;

if (temp == NULL) {

throw exception("Элемента не существует!");

}

}

if (index == 0) {

temp = hashTable[hashCode];

hashTable[hashCode] = hashTable[hashCode]->next;

delete temp;

}

else {

HashNode\* prev = hashTable[hashCode];

for (int i = 0; i < index - 1; i++)

{

prev = prev->next;

}

HashNode\* deleteNode = prev->next;

prev->next = deleteNode->next;

delete deleteNode;

}

throw exception("Элемента удалён!");

}

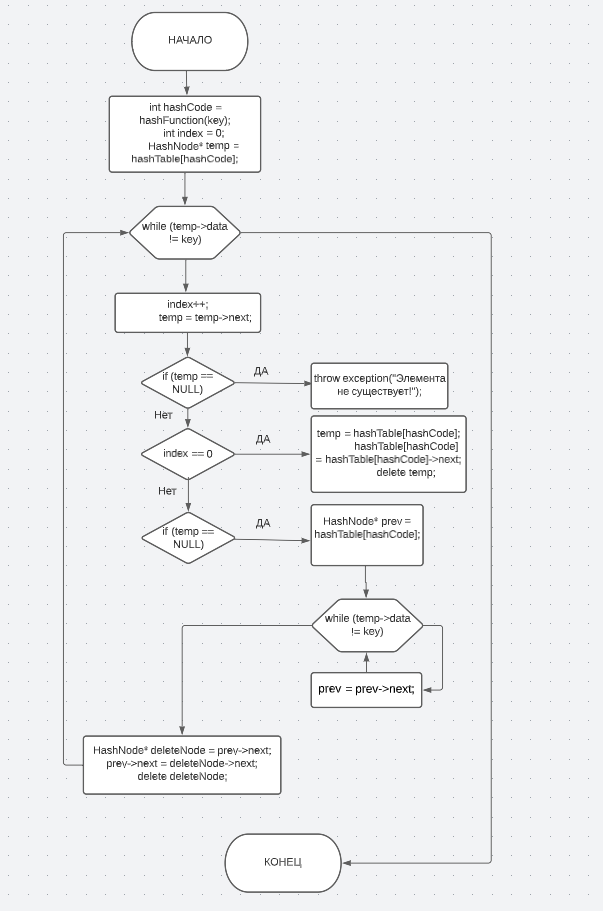


Рисунок 2.6.1 - Видалення елементу

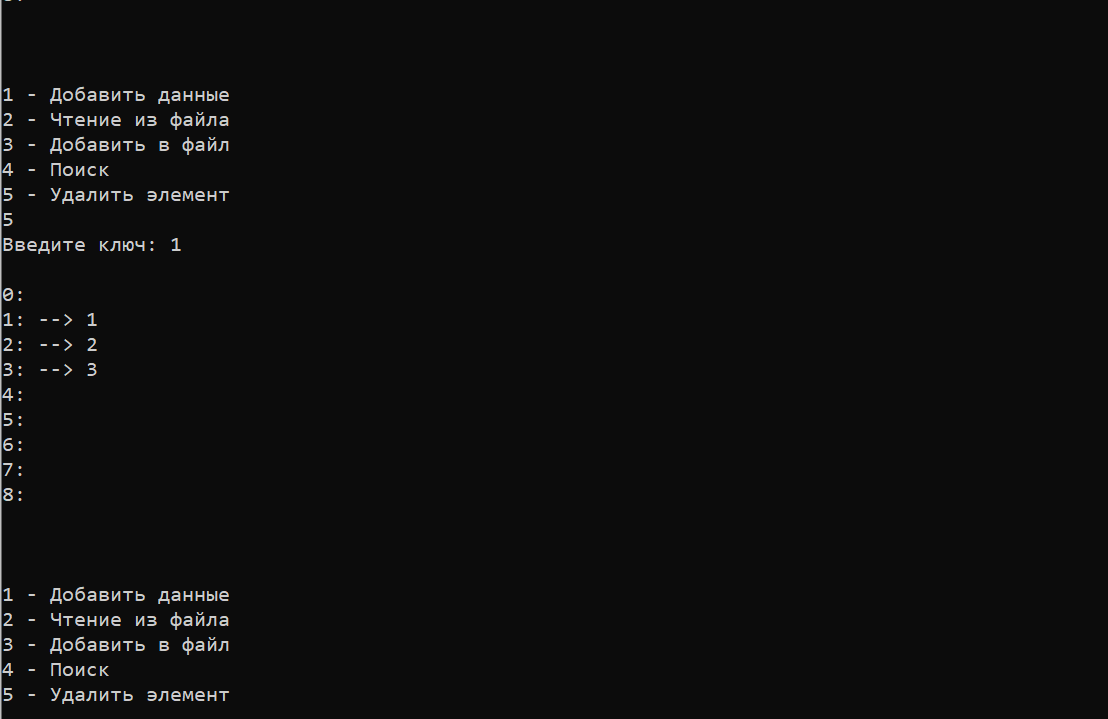


Рисунок 2.6.2 - Вигляд з консолі

2.7 Знайти число

Приклад коду:

int HTChain::searchElement(int key)

{

int hashCode = hashFunction(key);

int index = 0;

HashNode\* tempNode = hashTable[hashCode];

if (tempNode == NULL) {

throw exception("Элемент не найден!");

}

while (tempNode->data != key)

{

index++;

tempNode = tempNode->next;

if (tempNode == NULL) {

throw exception("Элемент не найден!");

}

}

cout << "Элемент найден в строке: " << hashCode

<< " Индекс: " << index;

}

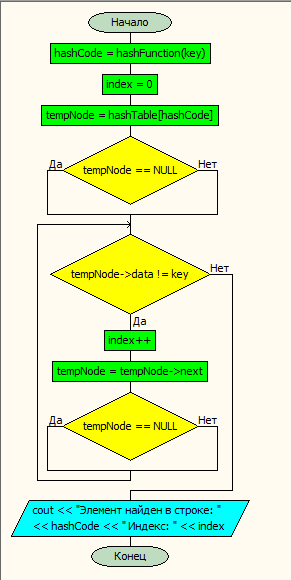


Рисунок 2.7.1 - Пошук елементу

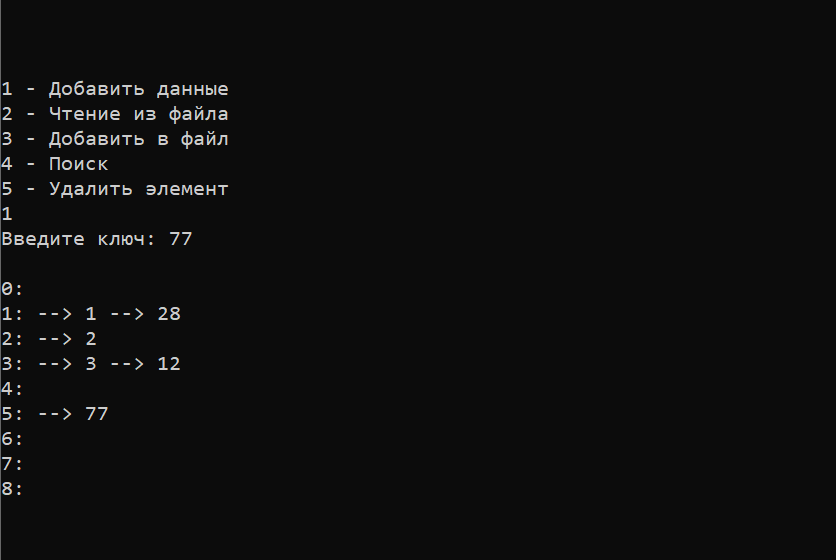


Рисунок 2.7.2 - вигляд з консолі

3 ВІДКРИТА АДРЕСАЦІЯ

3.1 Створення хеш-функції

Приклад коду:

int hashFunction(int key) {

return fmod(key, SIZE);

}

3.2 Створення хеш таблиці

Приклад коду :

HashTable\_OA::HashTable\_OA()

{

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

{

table[i].data = 0;

table[i].empty = 1;

}

}

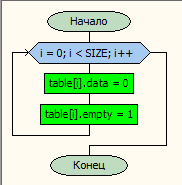


Рисунок 3.2.1 - заповнення хеш-таблиці

3.3 Лінійний метод

Приклад коду:

int HashTable\_OA::linearMethod(int key)

{

int hashCode = hashFunction(key);

int index = hashCode;

int count = 0;

while (count < SIZE)

{

if (table[index].empty)

return index;

index = (index + 1) % SIZE;

count++;

}

return -1;

}

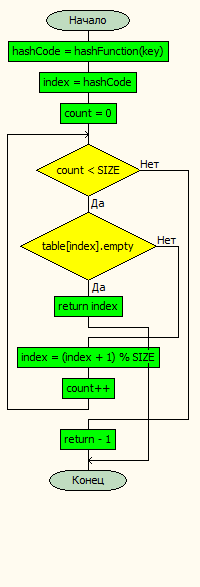


Рисунок 3.3.1 - Лінійний метод

3.3.1 Додавання елементу

Приклад коду:

void HashTable\_OA::addLinear(int key)

{

int position = linearMethod(key);

table[position].data = key;

table[position].empty = 0;

}

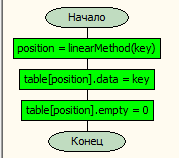


Рисунок 3.3.1.1. - Додавання елементу

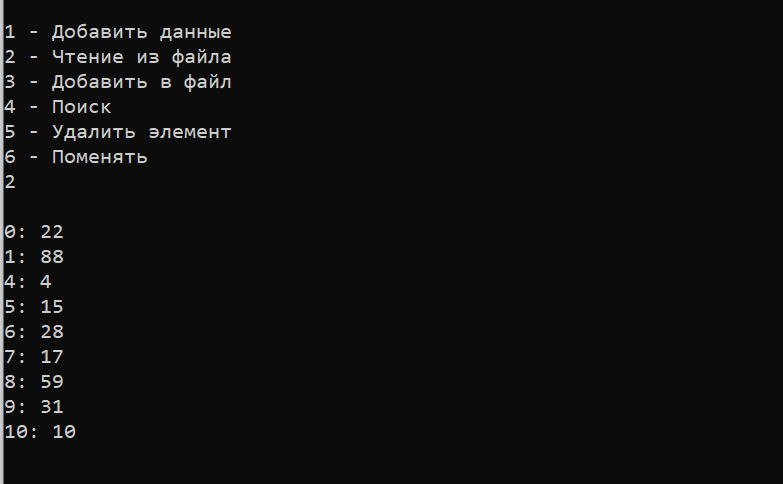


Рисунок 3.1.1.2 - вигляд з консолі

3.3.2 Видалення елементу

Приклад коду:

void HashTable\_OA::deleteLinear(int key)

{

int pos = searchLinear(key);

if (pos == -1) {

cout << "Элемент не найден" << endl;

}

else {

cout << "Элемент удалён" << endl;

table[pos].data = 0;

table[pos].empty = 1;

}

}

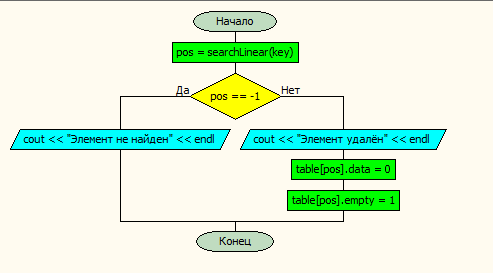


Рисунок 3.1.2.1. - Видалення елементу

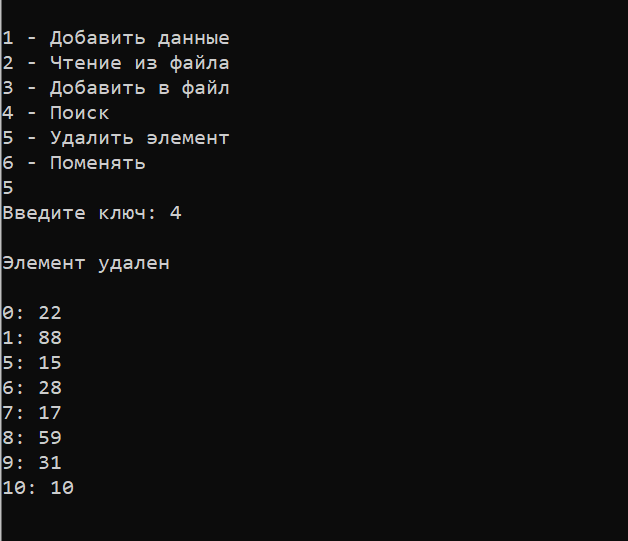
****

Рисунок 3.1.2.2 - вигляд з консолі

3.3.3 Пошук елементу

Приклад коду:

int HashTable\_OA::searchLinear(int key)

{

int hashCode = hashFunction(key);

while (table[hashCode].empty != 1)

{

if (table[hashCode].data == key)

return hashCode;

hashCode = (hashCode + 1) % SIZE;

}

return -1;

}

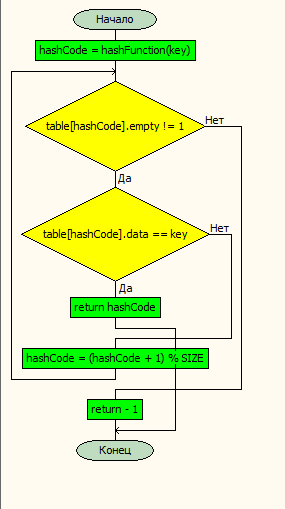


Рисунок 3.1.3.1. - Пошук елементу

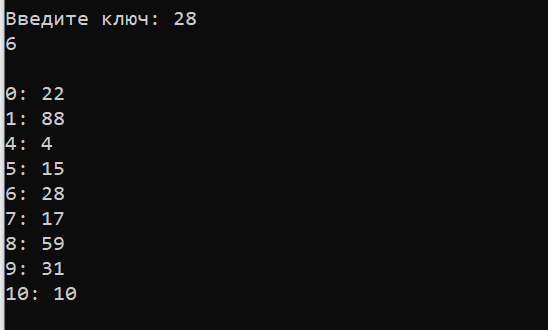
****

Рисунок 3.1.2.3 - вигляд з консолі

3.4 Квадратичний метод

Приклад коду:

int HashTable\_OA::quadraticMethod(int key)

{

int m1 = 1, m2 = 3;

int hashCode = hashFunction(key);

int index = 0;

int i = 0;

while (i < SIZE)

{

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

{

index = (hashCode + m1 \* i + m2 \* i \* i) % SIZE;

if (table[index].empty) {

return index;

}

}

hashCode++;

i++;

}

return -1;

}

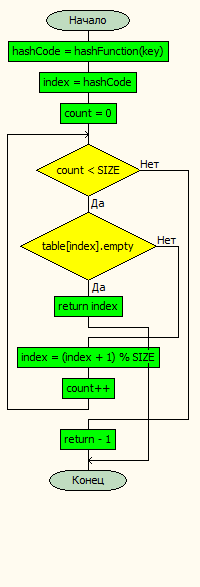


Рисунок 3.4.1 - Квадратичний метод

3.4.1 Додавання елементу

Приклад коду:

void HashTable\_OA::addQuadratic(int key)

{

int position = quadraticMethod(key);

table[position].data = key;

table[position].empty = 0;

}

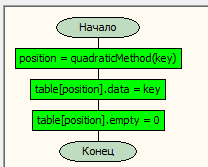


Рисунок 3.4.1.1. - Додавання елементу

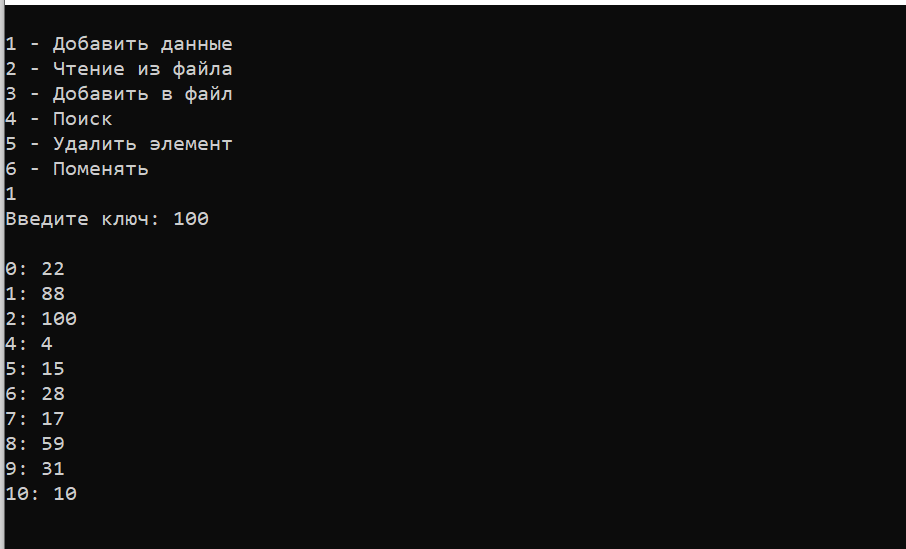


Рисунок 3.1.1.2 - вигляд з консолі

3.4.2 Видалення елементу

Приклад коду:

void HashTable\_OA::deleteQuadratic(int key)

{

int pos = searchQuadratic(key);

if (pos == -1) {

cout << "Элемент не найден" << endl;

}

else {

cout << "Элемент удалён" << endl;

table[pos].data = 0;

table[pos].empty = 1;

}

}

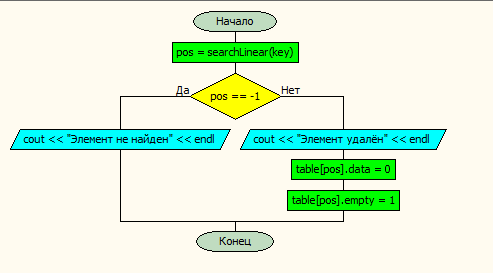


Рисунок 3.4.2.1. - Видалення елементу

****

Рисунок 3.1.2.2 - вигляд з консолі

3.4.3 Пошук елементу

Приклад коду:

int HashTable\_OA::searchQuadratic(int key)

{

int m1 = 1, m2 = 3;

int hashCode = hashFunction(key);

int index = 0;

int i = 0;

while (i < SIZE)

{

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

{

index = (hashCode + m1 \* i + m2 \* i \* i) % SIZE;

if (table[index].data == key) {

return index;

}

}

hashCode++;

i++;

}

return -1;

}

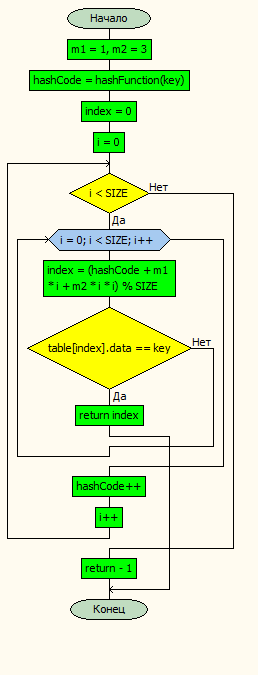


Рисунок 3.1.3.1. - Пошук елементу

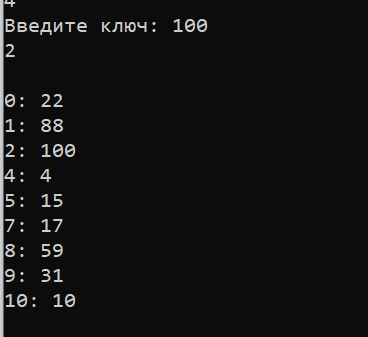
****

Рисунок 3.1.2.3 - вигляд з консолі

3.5.1 Подвійне хешування

Приклад коду:

int HashTable\_OA::doubleMethod(int key)

{

int hashCode = hashFunction(key);

int secHashCode = 1 + (key % (SIZE - 1));

int count = 0;

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

{

count = (hashCode + secHashCode + i) % SIZE;

if (table[count].empty) {

return count;

}

}

return -1;

}

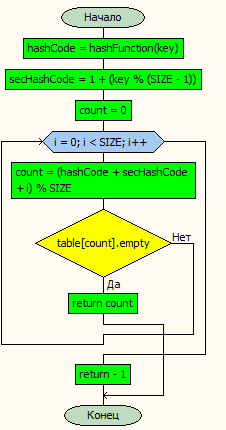


Рисунок 3.4.1 - Подвійне хешування метод

3.5.1 Додавання елементу

Приклад коду:

void HashTable\_OA::addDouble(int key)

{

int position = doubleMethod(key);

table[position].data = key;

table[position].empty = 0;

}

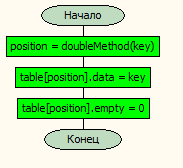


Рисунок 3.4.1.1. - Додавання елементу

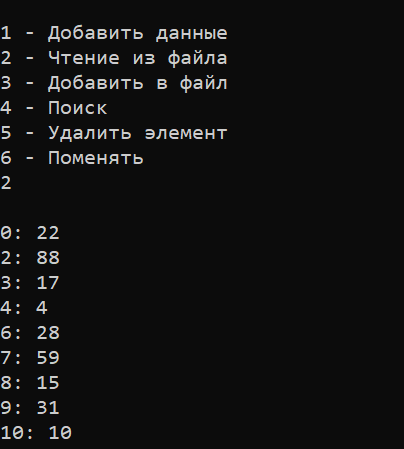


Рисунок 3.1.1.2 - вигляд з консолі

3.5.2 Видалення елементу

Приклад коду:

void HashTable\_OA::deleteDouble(int key)

{

int pos = searchDouble(key);

if (pos == -1) {

cout << "Элемент не найден" << endl;

}

else {

cout << "Элемент удалён" << endl;

table[pos].data = 0;

table[pos].empty = 1;

}

}

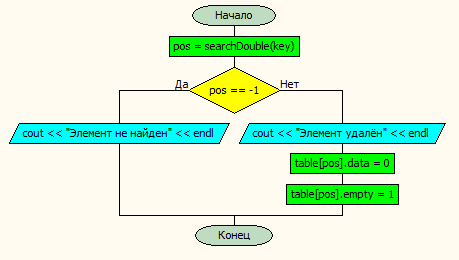


Рисунок 3.4.2.1. - Видалення елементу

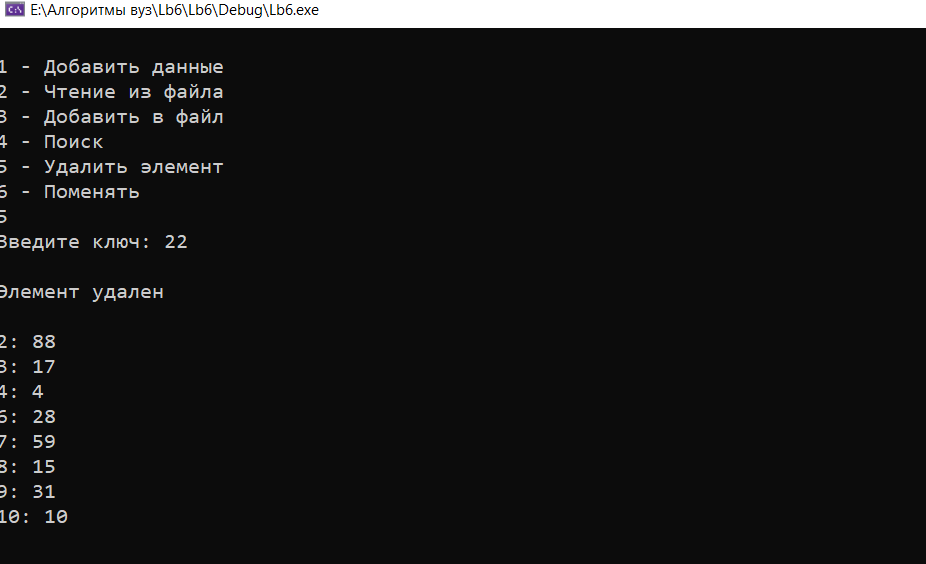
****

Рисунок 3.1.2.2 - вигляд з консолі

3.5.3 Пошук елементу

Приклад коду:

int HashTable\_OA::searchDouble(int key)

{

int hashCode = hashFunction(key);

int secHashCode = 1 + (key % (SIZE - 1));

int count = 0;

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

{

count = (hashCode + secHashCode + i) % SIZE;

if (table[count].data == key) {

return count;

}

}

return -1;

}

void HashTable\_OA::printTable()

{

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

{

cout << i << ": ";

if (table[i].empty == 0) {

cout << " -> " << table[i].data;

}

cout << endl;

}

}

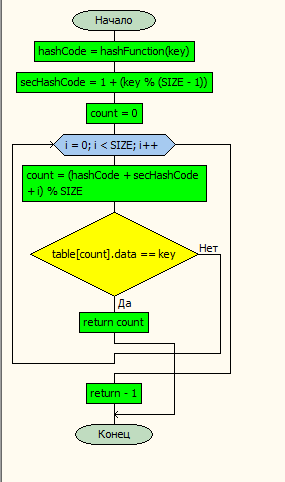


Рисунок 3.1.3.1. - Пошук елементу

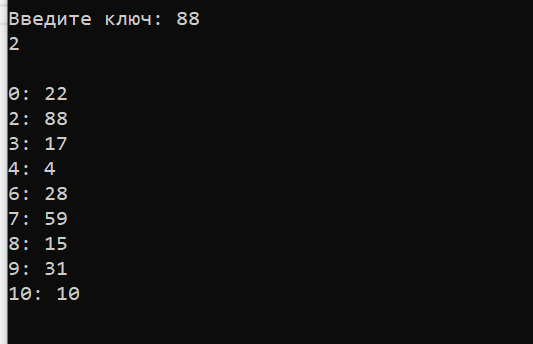
****

Рисунок 3.1.2.3 - вигляд з консолі

4 АСИМТОТИЧНИЙ АНАЛІЗ

Аналіз хеш-таблиці з прямою адресацією

Так як даний вид хеш-таблиці передбачає те що в дану таблицю будуть записані лише такі дані котрі не будуть викликати колізію то тимчасова складність для функції додавання, пошуку, видалення у всіх випадках буде – .

Аналіз хеш-таблиці з ланцюжками

Для функцій додавання, пошуку та видалення складність виглядає наступним чином:

* Найгірший випадок – ;
* Найкращий випадок – ;
* Середній випадок – , де n – кількість ключів котрі будуть вставлені в таблицю, m – розмір масиву таблиці;

Аналіз хеш-таблиці з відкритою адресацією

Лінійне пробування

Для функцій додавання, пошуку та видалення складність виглядає наступним чином:

* Найгірший випадок – ;
* Найкращий випадок – ;

– Середній випадок – , де n – кількість ключів котрі будуть вставлені в таблицю, m – розмір масиву таблиці;

Квадратичне пробування

Для функцій додавання, пошуку та видалення складність виглядає наступним чином:

* Найгірший випадок – ;
* Найкращий випадок – ;

– Середній випадок – , де n – кількість ключів котрі будуть вставлені в таблицю, m – розмір масиву таблиці;

Подвійне хешування

Для функцій додавання, пошуку та видалення складність виглядає наступним чином:

* Найгірший випадок – ;
* Найкращий випадок – ;

– Середній випадок – , де n – кількість ключів котрі будуть вставлені в таблицю, m – розмір масиву таблиці;

ВИСНОВКИ

Хеш-таблиця – це структура даних, що реалізує інтерфейс асоціативного масиву, тобто вона дозволяє зберігати пари виду "ключ-значення" і виконувати три операції: операцію додавання нової пари, операцію пошуку і операцію видалення пари по ключу.

Хеш-таблиця з прямою адресацією – це хеш-таблиці, котрі використовують ін’єктивні хеш-функції і не потребуються у механізмі вирішення колізії. Дані хеш-таблиці є найшвидшими так як у будь-якій операції мають наймінімальнішу тимчасову складність, але використовуються лише у тому випадку коли точно відомо, що в даній таблиці будуть зберігатися дані котрі не будуть викликати колізій.

Хеш-таблиця з використанням методу ланцюжків – це така хеш-таблиця в котрій використовується технологія вирішення колізій, котра полягає в тому, що елементи сукупності з рівними хеш-значеннями зв’язуються в ланцюжок-список. Головною перевагою такої хеш-таблиці є те що вона ніколи не переповниться так як список це динамічна структура даних. Головним недоліком такої таблиці є те що при малому розмірі масиву таблиці, постане проблема в довжині ланцюжків із-за чого буде доволі довгий пошук необхідного елементу. Але при великому розмірі масиву таблиці та при мінімальних довжинах ланцюгів можливо добитися мінімальної тимчасової складності як і в хеш-таблиці з прямою адресацією.

Хеш-таблиця з відкритою адресацією – це хеш-таблиця в якій використовується технологія вирішення колізій, котра передбачає зберігання записів в самій хеш-таблиці. Для вирішення колізій в даній хеш-таблиці використовується методи повторного хешування.

Лінійне пробування – елементи хеш-таблиці послідовно проглядаються з деяким фіксованим інтервалом k між елементами. Недоліком даного виду пробування є кластеризація.

Квадратичне пробування – це таке пробування коли інтервал між елементами з кожним кроком збільшується на константу. Завдяки нелінійності такої адресації зменшується число проб при більшій кількості ключів-синонімів. Але мінусом такого пробування є те що навіть відносно невелике число проб може швидко привести до виходу за адресний простір невеликої таблиці внаслідок квадратичної залежності адреси від номера спроби.

Подвійне хешування – засноване на нелінійній адресації, що досягається за рахунок підсумовування значень основної і додаткової хеш-функції. Завдяки такому підходу знижується ефект кластеризації.

Хеш-таблицю з відкритою адресацією доцільно використовувати коли немає можливості витрачати пам’ять на зберігання вказівників списку, або якщо немає можливості зберігати елементи з зовні таблиці. Мінусами такої таблиці є те що вона може закінчитися, а також при більшому заповненні збільшується і тимчасова складність.