Міністерство освіти і науки України

Національний технічний університет України

«Київський політехнічний інститут ім. Ігоря Сікорського»

Інститут прикладного системного аналізу

Кафедра системного проектування

Алгоритми та структури даних

Лабораторна робота №2

“Дослідження структури даних хеш-таблиця”

Аносов Ілля

ДА-21

Київ

2023

**Мета роботи:**

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

**ВАРІАНТ 1**

**Завдання**

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

**1.** Реалізувати функцію generateRandLong():

**1.1** Функція повинна генерувати та повертати випадкове велике число, що буде використовуватись як унікальний ключ-ідентифікатор даних у хеш-таблиці. Сигнатура функції: long long generateRandLong().

**2.** Реалізувати статичну 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, виділити її в окремі header та cpp файли.

**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) – знайти хеш-функцію від ключа
* insert(key, value) – додати значення під відповідним ключем (заміна у випадку існування ключа)
* find(key) – знайти значення під відповідним ключем, функція повертає вказівник на знайдений елемент або nullptr
* erase(key) – видалити значення під відповідним ключем
* size() – знайти кількість елементів в хеш-таблиці

**3.** Додати динамічне розширення хеш-таблиці:

**3.1** На кожному кроці обчислювати loadFactor (поточну завантаженість таблиці) та змінювати розмір масиву bucketsArray в залежності від maxLoadFactor (максимально можлива завантаженість таблиці).

**3.2** Початковий розмір масиву bucketsArray встановити невеликим, при перевищенні значення maxLoadFactor динамічно збільшувати розмір.

**3.3** При перевиділенні пам’яті для всіх елементів хеш-таблиці необхідно обчислити нові позиції, відповідно до зміни хеш-функції.

**4.** Провести тестування хеш-таблиці:

**4.1** Використати декілька різних значень maxLoadFactor (наприклад 9, 4 та 0.5), використати декілька різних хеш-функцій.

**4.2** За допомогою вказаної нижче функції testHashTable() провести тестування із замірами часу та обрати оптимальні хеш-функцію та значення maxLoadFactor.

**4.3** Дослідити використання готового бібліотечного рішенням STL unordered\_map, порівняти отримані результати зі своїм рішенням.

Зображення, що містить текст

Автоматично згенерований опис

**headhash.h**

#pragma once

#include <iostream>

#define loadFactor 0.4

#define koef 5

using namespace std;

struct Node {

Node() {

asymptotic = 0;

name = "-";

usefulness = "-";

}

Node(int anAsymptotic, string aName, string aUsefulness):

asymptotic(anAsymptotic),

name(aName),

usefulness(aUsefulness) {}

int asymptotic;

string name;

string usefulness;

};

struct HashNode {

long long key;

Node node;

HashNode\* next;

};

struct LinkedList {

private:

HashNode\* head = nullptr;

public:

void push\_front(long long key, Node list);

HashNode\* get(long long key);

HashNode\* pop\_front();

int size();

bool remove(long long key);

};

struct HashTable {

private:

int N = 1000;

int hashSize = 0;

LinkedList\* bucketsArray = new LinkedList[N];

int hash(long long key);

public:

void insert(long long key, Node node);

Node\* find(long long key);

void erase(long long key);

int size();

};

**HashTable.cpp**

#include "headhash.h"

int HashTable::hash(long long key) {

key = key % N;

return key;

}

void HashTable::insert(long long key, Node node) {

if ((double)size() / N >= loadFactor) {

N \*= koef;

LinkedList\* newList = new LinkedList[N];

for (int i = 0; i < N / koef; i++) {

bool isEmpty = false;

while (!isEmpty) {

HashNode\* savedNode = bucketsArray[i].pop\_front();

if (savedNode) {

int bucketNumber = hash(savedNode->key);

HashNode\* change = newList[bucketNumber].get(savedNode->key);

if (change) {

change->key = savedNode->key;

change->node = savedNode->node;

}

else

newList[bucketNumber].push\_front(savedNode->key, savedNode->node);

delete savedNode;

}

else

isEmpty = true;

}

}

delete[] bucketsArray;

bucketsArray = newList;

}

int bucketNumber = hash(key);

Node\* change = find(key);

if (change) {

\*change = node;

return;

}

bucketsArray[bucketNumber].push\_front(key, node);

hashSize++;

}

Node\* HashTable::find(long long key) {

int bucketNumber = hash(key);

HashNode\* returnNode = bucketsArray[bucketNumber].get(key);

if(returnNode)

return &bucketsArray[bucketNumber].get(key)->node;

return nullptr;

}

void HashTable::erase(long long key) {

int bucketNumber = hash(key);

if(bucketsArray[bucketNumber].remove(key))

hashSize--;

}

int HashTable::size() {

return hashSize;

}

**LinkedList.cpp**

#include "headhash.h"

void LinkedList::push\_front(long long key, Node node) {

HashNode\* newNode = new HashNode();

newNode->key = key;

newNode->node = node;

newNode->next = head;

head = newNode;

}

HashNode\* LinkedList::get(long long key) {

HashNode\* getNode = head;

if (!head)

return nullptr;

while (getNode) {

if (getNode->key == key)

return getNode;

getNode = getNode->next;

}

return nullptr;

}

HashNode\* LinkedList::pop\_front() {

if (!head)

return nullptr;

HashNode\* save\_head = head;

head = head->next;

return save\_head;

}

int LinkedList::size() {

HashNode\* checkNode = head;

int current\_size = 0;

while (checkNode) {

checkNode = checkNode->next;

current\_size++;

}

return current\_size;

}

bool LinkedList::remove(long long key) {

HashNode\* deleteNode = head;

HashNode\* link = nullptr;

int counter = 0;

while (deleteNode) {

if (deleteNode->key == key) {

if (link)

link->next = deleteNode->next;

else

head = head->next;

delete deleteNode;

return true;

}

link = deleteNode;

deleteNode = deleteNode->next;

}

return false;

}

**main.cpp**

#include <unordered\_map>

#include "headhash.h"

#include <iostream>

#include <ctime>

long long generateRandLong() {

long long number = 0;

int lengthNumber = rand() % 17;

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

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

}

return number;

}

bool testHashTable(){

...

}

int main()

{

srand(time(0));

testHashTable();

}

**Висновок:**

В результаті виконання лабораторної роботи було реалізовано статичну та динамічну хеш-таблицю за методом ланцюжків. Для її роботи використано методи зв’язних списків та основані на них функції хеш-таблиці. Порівняно створену структуру з бібліотечним рішенням за різних значень maxLoadFactor та коефіцієнту збільшення масиву для динамічного варіанту. При значені maxLoadFactor=0.4 та розширені масиву у 2 рази час додавання, видалення та знаходження по 500 000 елементів приблизно рівні. Але, зі збільшенням коефіцієнту змінення масиву до 5, витрачений час значно зменшується, і для тієї самої кількості елементів різниця в середньому складає близько 0.7 секунди.