**Санкт-Петербургский национальный исследовательский университет**

**информационных технологий, механики и оптики**

**Кафедра информатики и прикладной математики**

**Отчет по лабораторной работе №1**

**«Хэш-таблицы»**

**Вариант 3**

Выполнил: Гхази Даниэль

Группа P3218

Преподаватель: Зинчик А.А.

г. Санкт-Петербург

2017 г.

**Текст Задания**

Во всех вариантах требуется разработать программу, реализующую комбинированный способ организации таблицы идентификаторов. Для организации таблицы используется простейшая хэш-функция, указанная в варианте задания, а при возникновении коллизий используется дополнительный метод размещения идентификаторов в памяти. Если в качестве этого метода используется дерево или список, то они должны быть связаны с элементом главной хэш-таблицы.

В каждом варианте требуется, чтобы программа сообщала среднее число коллизий и среднее количество сравнений, выполненных для поиска идентификатора.

|  |  |  |
| --- | --- | --- |
| 3. | Сумма кодов первых трех букв | Метод цепочек |

**Текст Программы**

#include <iostream>

#include <vector>

#include <fstream>

#include <regex>

#include "main.h"

int main()

{

std::vector<ElementOfHashTable> hashTable;

hashTable.reserve(constants::reserevedMemory);

bool exit = false;

MenuItems menuItem = EXIT;

int numberOfCollisions = 0;

int numberOfElementsGivenForInput = 0;

int numberOfSearchesMadeByUser = 0;

int numberOfComparisons = 0;

bool hashTableExists = false;

showMenu();

while (!exit)

{

menuItem = getAndCheckMenuItem();

doChosenAction(&exit, menuItem, hashTable, &numberOfCollisions,

&numberOfElementsGivenForInput, &numberOfSearchesMadeByUser,

&numberOfComparisons, &hashTableExists);

}

return 0;

}

void showMenu()

{

std::cout << "1. Input from file\n2. Search\n3. Show stats\n4. Exit\n\n";

}

MenuItems getAndCheckMenuItem()

{

std::string menuItemPrekeeper = "";

MenuItems menuItem = EXIT;

std::regex menuItemChecker(constants::menuRegex);

while (true)

{

getline(std::cin, menuItemPrekeeper);

bool validInput = std::regex\_match(menuItemPrekeeper, menuItemChecker);

if (!validInput)

{

std::cout << ("Invalid input.\n\n");

continue;

}

else

{

menuItem = MenuItems(menuItemPrekeeper[0] - '0');

return menuItem;

}

}

}

void doChosenAction(bool \*exit, MenuItems menuItem, std::vector<ElementOfHashTable> &hashTable,

int\* numberOfCollisions, int\* numberOfElementsGivenForInput, int\* numberOfSearchesMadeByUser,

int\* numberOfComparisons, bool\* hashTableExists)

{

switch (menuItem)

{

case INPUT:

inputFromFile(hashTable, numberOfCollisions, numberOfElementsGivenForInput);

\*hashTableExists = true;

break;

case SEARCH:

if (\*hashTableExists)

{

searchElement(hashTable, numberOfComparisons);

(\*numberOfSearchesMadeByUser)++;

}

else

{

std::cout << "Hash table was not created.\n\n";

}

break;

case STATS:

showStats(numberOfCollisions, numberOfElementsGivenForInput,

numberOfSearchesMadeByUser, numberOfComparisons);

break;

case EXIT:

\*exit = true;

break;

}

}

void inputFromFile(std::vector<ElementOfHashTable> &hashTable,

int\* numberOfCollisions, int\* numberOfElementsGivenForInput)

{

hashTable.clear();

hashTable.resize(constants::baseSizeOfHashTable \* constants::startingExtentionCoefficientOfHashTable);

ElementOfHashTable\* freePtr = &hashTable[constants::baseSizeOfHashTable];

(\*numberOfCollisions) = 0;

(\*numberOfElementsGivenForInput) = 0;

std::string fileName = "";

std::cout << "Input file name: ";

getline(std::cin, fileName);

std::ifstream inf(fileName);

if (!inf)

{

std::cout << "File not found or failed to open it.\n\n";

return;

}

std::string newElement;

while (inf >> newElement)

{

newElement = processNewElement(newElement);

if (newElement != constants::noLettersInWordMarker)

{

insertNewElement(hashTable, &freePtr, newElement, numberOfCollisions, numberOfElementsGivenForInput);

}

}

std::cout << "Success!\n\n";

}

std::string processNewElement(std::string newElement)

{

std::string processedElement = "";

processedElement.resize(constants::maxLengthOfInputWord, ' ');

int indexToPutNewLetter = 0;

if (newElement.size() > processedElement.size()) newElement.resize(processedElement.size());

for (int i = 0; i < newElement.size(); i++)

{

newElement[i] = tolower(newElement[i]);

if ((newElement[i] >= constants::asciiCodeOfLetter\_a) &&

(newElement[i] <= constants::asciiCodeOfLetter\_z))

{

processedElement[indexToPutNewLetter++] = newElement[i];

}

}

// No letters in word, hence do not add it to hash-table.

if (indexToPutNewLetter == 0) return constants::noLettersInWordMarker;

return processedElement;

}

void insertNewElement(std::vector<ElementOfHashTable> &hashTable, ElementOfHashTable\*\* freePtr,

std::string newElement, int\* numberOfCollisions, int\* numberOfElementsGivenForInput)

{

int hashCode = hashFunction(newElement);

ElementOfHashTable\* slotPtr = &hashTable[hashCode];

(\*numberOfElementsGivenForInput)++;

while (true)

{

if (newElement == slotPtr->value) return;

else if (slotPtr->value == "")

{

slotPtr->value = newElement;

break;

}

else if (slotPtr->referenceToCollidedElement == NULL)

{

slotPtr->referenceToCollidedElement = \*freePtr;

(\*freePtr)->value = newElement;

(\*freePtr)++;

(\*numberOfCollisions)++;

break;

}

else

{

slotPtr = slotPtr->referenceToCollidedElement;

(\*numberOfCollisions)++;

continue;

}

}

// Checking if hashtable is full. If it is, get control of already allocated slots.

if ((\*freePtr) == &hashTable[hashTable.size() - 1])

{

hashTable.resize(hashTable.size() \* constants::regularExtentionCoefficientOfHashTable);

}

}

// Sum of codes of first 3 letters.

int hashFunction(std::string element)

{

int bias = constants::biasForEachSymbolToBeginningOfHashTable;

int hashCode = element[0] - bias;

for (int i = 1; i < 3; i++)

{

if (element[i] != ' ') hashCode += element[i] - bias;

else break;

}

return hashCode;

}

void searchElement(std::vector<ElementOfHashTable> &hashTable, int\* numberOfComparisons)

{

std::string valueToSearch = "";

std::regex wordToSearchChecker(constants::searchInputRegex);

std::cout << "Input value to search: ";

std::getline(std::cin, valueToSearch);

bool validInput = std::regex\_match(valueToSearch, wordToSearchChecker);

if (!validInput)

{

std::cout << "Not found!\n\n";

return;

}

valueToSearch.resize(constants::maxLengthOfInputWord, ' ');

int hashCode = hashFunction(valueToSearch);

ElementOfHashTable\* slotPtr = &hashTable[hashCode];

while (true)

{

(\*numberOfComparisons)++;

if (valueToSearch == slotPtr->value)

{

std::cout << "Found!\n\n";

return;

}

else if (slotPtr->referenceToCollidedElement != NULL)

{

slotPtr = slotPtr->referenceToCollidedElement;

continue;

}

else

{

std::cout << "Not found!\n\n";

return;

}

}

}

void showStats(int\* numberOfCollisions, int\* numberOfElementsGivenForInput,

int\* numberOfSearchesMadeByUser, int\* numberOfComparisons)

{

double averageNumberOfCollisions = 0;

double d\_NumberOfCollisions = \*numberOfCollisions;

if (\*numberOfElementsGivenForInput != 0)

{

averageNumberOfCollisions = d\_NumberOfCollisions / (\*numberOfElementsGivenForInput);

}

double averageNumberOfComparisonsMadeDuringSearch = 0;

double d\_NumberOfComparisons = \*numberOfComparisons;

if (\*numberOfSearchesMadeByUser != 0)

{

averageNumberOfComparisonsMadeDuringSearch = d\_NumberOfComparisons / (\*numberOfSearchesMadeByUser);

}

std::cout << "Average number of collisions = " << averageNumberOfCollisions << std::endl;

std::cout << "Average number of comaparisons made during search = " << averageNumberOfComparisonsMadeDuringSearch

<< std::endl << std::endl;

}

**Вывод**

В результате выполнения лабораторной мною были изучены различные способы организации хэш-таблиц и разрешения коллизий. Применения метода цепочек для разрешения коллизий позволяет сохранить высокую степень заполняемости хэш-таблицы.