Отчёт по лабораторной работе

**Таблицы**

**Выполнила:**

студент ИИТММ гр. 382003-1

Полякова А.В.

Нижний Новгород

2022 г.

Содержание

[Введение 3](#_Toc532661930)

[Постановка задачи 4](#_Toc532661931)

[Руководство пользователя 5](#_Toc532661932)

[Руководство программиста 6](#_Toc532661933)

[Описание структуры программы 6](#_Toc532661934)

[Описание структур данных 8](#_Toc532661935)

[Описание алгоритмов 9](#_Toc532661936)

[Заключение](#_Toc532661937) 10

[Приложения](#_Toc532661939) 11

[Приложение 1 (класс TableElement) 1](#_Toc532661940)1

[Приложение 2 (класс UnorderedTable) 1](#_Toc532661941)2

[Приложение 3 (класс OrderedTable) 1](#_Toc532661942)3

[Приложение 4 (класс TreeNode) 1](#_Toc532661943)4

[Приложение 5 (класс Tree) 1](#_Toc532661944)5

[Приложение 6 (класс HashNode) 1](#_Toc532661945)7

[Приложение 7 (класс HashTable)](#_Toc532661946) [1](#_Toc532661946)8

Введение

Существуют задачи, в которых есть необходимость использовать табличные структуры данных. Их характерные черты состоят в том, что они хранят данные в виде пары «ключ-значение» - записи в таблице, позволяют выполнять вставку записей, поиск и удаление по ключу. Как правило, ключ предполагается уникальным для каждой записи.

Способы представления таблиц бывают разные и определяются характером решаемой задачи – например, где-то поиск происходит намного чаще, чем вставка и удаление, а где-то наоборот. В лабораторной работе рассматриваются реализации неупорядоченной, упорядоченной и хеш-таблиц.

Постановка задачи

Требуется разработать программу, которая предоставляет возможность хранить данные в таблицах 3-х типов:

1)Неупорядоченная таблицы

2)Упорядоченная таблица

3)Хэш-таблица

В качестве данных должны использоваться полиномы из лабораторной работы 4.

Работа происходит сразу с таблицами всех типов.

Пользователь может положить в таблицу свой полином, с заданием ему некоторого наименования (ключа).

Пользователь может запросить (найти в/извлечь из) таблицы полином с заданным наименованием.

В ходе выполнения программы у пользователя должна оставаться возможность производить операции с полиномами, а также размещать полученный результат в таблицу.

Во время работы программы происходит логирование количества произведенных операций. Их число для каждой таблицы выводится на экран или в файл после выполнения соответствующей операции.

Руководство пользователя

* При запуске программы выводятся правила использования программы и корректного ввода данных.
* Вы можете ввести «exit», чтобы закрыть программу, «get», чтобы получить многочлен, находящийся в таблице, или сам многочлен – правила его ввода отображаются при запуске программы.
* При выборе варианта «get» введите имя многочлена в таблице. Если он есть, выведется его значения, и можно будет выполнить операции, если нет – вы увидите сообщение об ошибке.
* Если ввод некорректен, произойдёт возврат к выбору действия.
* Когда многочлен получен, можно выполнить следующие действия: выйти и программы, сложить, вычесть или умножить на многочлен, умножить на число или сохранить многочлен.
* Для сохранения нужно будет ввести ключ - имя многочлена. Оно должно быть уникальным, т.е. в таблице не должен содержаться многочлен с таким же именем. Если же он есть, нужно снова ввести имя, и так до тех пор, пока оно не будет уникальным.
* Для выполнения сложения, вычитания или умножения на многочлен нужно ввести многочлен, для умножения на число – число. Ввод многочлена происходит так же, как в начале работы программы, ввод числа происходит в соответствии с форматом ввода вещественных чисел в C++.
* Как только арифметическая операция выполнится, выведется её результат. Полученный многочлен можно добавить или не добавить в таблицу.
* Если выбран первый вариант, требуется дать имя уникальное имя многочлену.
* Далее происходит возврат к исходному окну, где вы можете выйти, взять многочлен из таблицы или ввести его вручную.

Руководство программиста

## Описание структуры программы

**Библиотеки, использующиеся в программе:**

1. iostream (Для ввода и вывода информации)
2. vector (Для временного хранения слагаемых многочлена)
3. sstream (Для удобной работы со строками)
4. utility (Содержит различные инструменты C++)

**Алгоритм программы:**

1. **Пользователь может ввести слова «exit», «get» или многочлен по правилам, выводимым на экран при начале работы программы.**
2. **Ввод «exit» приводит к закрытию программы.**
3. **Ввод «get» позволяет получить многочлен из таблицы по ключу. Пользователь должен ввести ключ – имя многочлена. По ключу производится поиск в таблице; если запись с ним есть, в консоль выводятся многочлен и количество операций при поиске в каждой таблице, дальнейшие действия – на шаге 5, иначе выводится сообщение, что запись не найдена, и происходит возврат к шагу 1.**
4. **Если непосредственно вводить многочлен, при корректном вводе происходит его вывод в консоль и переход к следующему шагу, иначе – вывод сообщения об ошибке и возврат к шагу 1.**
5. **На этом шаге можно выйти из программы, выполнить арифметические операции над многочленом или сохранить его в таблицу.**

**При выборе сохранения нужно ввести уникальное имя. Если оно неуникально, придётся повторять попытку, пока имя не будет уникальным. Как только корректное имя введено, запись добавляется в таблицы, выводится количество операций при добавлении, происходит переход к шагу 1.**

**При выборе сложения, вычитания или умножения на многочлен нужно ввести второй многочлен аналогично шагу 1.**

**При выборе умножения на константу нужно ввести число. Процесс будет повторяться, пока ввод не будет корректен.**

**При выборе выхода из программы происходит завершение её работы.**

1. **По выполнении арифметических операций выводится результат и предлагается сохранить его в таблицу. Если решено сохранить, необходимо ввести уникальное имя. После этого идёт возврат к шагу 1.**

***Описание структур данных***

**Классы:**

* **TableElement** – запись в виде пары «ключ-значение», является элементом, хранящимся во всех таблицах.
* **UnorderedTable** – реализация неупорядоченной таблицы с помощью std::vector.
* **OrderedTable** – реализация упорядоченной таблицы с помощью двоичного дерева поиска.
* **TreeNode** – вершина двоичного дерева поиска.
* **Tree** – реализация двоичного дерева поиска.
* **HashTable** – реализация хеш-таблицы с использованием метода цепочек.
* **HashNode** – звено списка, хранящегося в каждой ячейке массива в хеш-таблице.

## Описание алгоритмов

Во всех таблицах набор операций одинаков: поиск и вставка.

**Неупорядоченная таблица:**

**Вставка:** добавление элемента в конец вектора.

**Поиск:** линейный поиск в векторе (прохождение по всем элементам).

**Упорядоченная таблица:**

**Вставка:** поиск нужного места в дереве и добавление новой записи в качестве новой записи в качестве потомка.

**Поиск:** начиная с корня, перемещаемся в левое поддерево, если искомый ключ меньше текущего, в правое – если больше текущего. Если значение совпадает, запись найдена.

**Хеш-таблица:**

**Вставка:** вычисляем хеш от ключа, проверяем, есть ли в списке запись с данным ключом. Если есть, заменяем значение, если нет – добавляем в конец списка новую запись.

**Поиск:** вычисляем хеш от ключа и ищем запись в списке.

Заключение

В лабораторной работе были разработаны классы неупорядоченных, упорядоченных и хеш-таблиц, а также вспомогательные классы для них. Их корректность проверена на тестах, созданных при помощи Google Tests. Также разработано пользовательское приложение, позволяющее совершать арифметические операции над многочленами и сохранять их с именами в таблице.

Приложения

## Приложение 1 (класс TableElement)

class TableElement

{

std::string \_key; // Ключ

Polynomial \_value; // Значение

public:

// Конструктор по умолчанию

TableElement(){}

// Конструктор с параметрами

TableElement(const std::string& key, const Polynomial& value)

{

\_key = key;

\_value = value;

}

// Получение ключа

std::string key() const

{

return \_key;

}

// Получение данных

Polynomial value() const

{

return \_value;

}

// Перегрузка оператора<

friend bool operator<(const TableElement& te1, const TableElement& te2)

{

return te1.key() < te2.key();

}

// Перегрузка оператора==

friend bool operator==(const TableElement& te1, const TableElement& te2)

{

return te1.key() == te2.key();

}

// Перегрузка оператора>

friend bool operator>(const TableElement& te1, const TableElement& te2)

{

return te1.key() > te2.key();

}

};

## Приложение 2 (класс UnorderedTable)

class UnorderedTable

{

std::vector<TableElement\*> table; // Массив

public:

// Конструктор таблицы

UnorderedTable()

{

table = std::vector<TableElement\*>();

}

bool existsKey(const std::string& key)

{

for (auto te : table)

if (te->key() == key)

return true;

return false;

}

// Добавление данных

void add(const std::string& key, const Polynomial& p)

{

table.push\_back(new TableElement(key, p));

std::cout << "Неупорядоченная таблица - операций добавления: 1." << std::endl;

}

// Добавление данных

void add(const TableElement& te)

{

add(te.key(), te.value());

}

// Получение данных

Polynomial get(const std::string& key)

{

for (size\_t i = 0; i < table.size(); i++)

{

if (table.at(i)->key() == key)

{

std::cout << "Неупорядоченная таблица - операций поиска: "

<< i + 1 << "." << std::endl;

return table.at(i)->value();

}

}

return NULL;

}

## };

## Приложение 3 (класс OrderedTable)

class OrderedTable

{

Tree \*table; // Бинарное дерево поиска

public:

// Конструктор таблицы

OrderedTable()

{

table = new Tree();

}

bool existsKey(const TableElement& te)

{

return table->existsKey(te);

}

// Добавление данных

void add(const std::string& key, const Polynomial& p)

{

int count = table->add(TableElement(key, p));

std::cout << "Упорядоченная таблица - операций добавления: "

<< count << std::endl;

}

// Добавление данных

void add(const TableElement& te)

{

add(te.key(), te.value());

}

// Поиск полинома

Polynomial get(const std::string& key)

{

auto result = table->find(TableElement(key, Polynomial()));

std::cout << "Упорядоченная таблица - операций поиска: "

<< result.second << "." << std::endl;

return result.first->data.value();

}

};

## Приложение 4 (класс TreeNode)

template<typename T>

struct TreeNode

{

T data; // Данные

TreeNode \*left, \*right; // Узлы-потомки

// Конструктор по умолчанию

TreeNode<T>()

{

this->data = NULL;

this->left = NULL;

this->right = NULL;

}

// Конструктор с параметрами

TreeNode<T>(const T& data)

{

this->data = data;

this->left = NULL;

this->right = NULL;

}

// Конструктор с параметрами

TreeNode<T>(const T& data, TreeNode<T> \*left, TreeNode<T> \*right)

{

this->data = data;

this->left = left;

this->right = right;

}

};

## Приложение 5 (класс Tree)

class Tree

{

TreeNode<TableElement> \*root; // Корень

// Добавление в дерево

int \_add(TreeNode<TableElement> \*current, const TableElement& data, int count)

{

// Если пуст, то создание нового узла

if (current == NULL)

{

current = new TreeNode<TableElement>(data);

return count + 1;

}

else if (current->data < data)

{

if (current->right == NULL)

{

current->right = new TreeNode<TableElement>(data);

return count + 1;

}

else

return \_add(current->right, data, count + 1);

}

else

{

if (current->left == NULL)

{

current->left = new TreeNode<TableElement>(data);

return count + 1;

}

else

return \_add(current->left, data, count + 1);

}

}

// Поиск по дереву

std::pair<TreeNode<TableElement>\*, int> \_find(TreeNode<TableElement> \*current,

const TableElement& key, int count)

{

// Если текущий пуст или нашли ключ, то возвращаем текущий

if (current == NULL || current->data == key)

return std::pair<TreeNode<TableElement>\*, int>(current, count + 1);

// Иначе переход в левое или правое поддерево

else if (current->data < key)

return \_find(current->right, key, count + 1);

else

return \_find(current->left, key, count + 1);

}

bool \_existsKey(TreeNode<TableElement> \*current, const TableElement& te)

{

if (current == NULL)

return false;

else if (current->data.key() == te.key())

return true;

else if (current->data < te)

return \_existsKey(current->right, te);

else

return \_existsKey(current->left, te);

}

public:

// Конструктор по умолчанию

Tree()

{

root = NULL;

}

// Конструктор с параметром

Tree(const TableElement& data)

{

root = new TreeNode<TableElement>(data);

}

// Добавление данных

int add(const TableElement& data)

{

// Если корень пуст, создаем его

if (root == NULL)

{

root = new TreeNode<TableElement>(data);

return 1;

}

else

return \_add(root, data, 0);

}

// Поиск по ключу

std::pair<TreeNode<TableElement>\*, int> find(const TableElement& key)

{

return \_find(root, key, 0);

}

bool existsKey(const TableElement& te)

{

return \_existsKey(root, te);

}

};

## Приложение 6 (класс HashNode)

template <typename K, typename V>

class HashNode

{

K \_key; // Ключ

V \_value; // Данные

HashNode \*\_next; // Следующий узел

public:

// Конструктор

HashNode(const K &key, const V &value)

{

\_key = key;

\_value = value;

\_next = NULL;

}

// Получение ключа

K key() const

{

return \_key;

}

// Получение данных

V value() const

{

return \_value;

}

// Получение следующего узла

HashNode\* next() const

{

return \_next;

}

// Изменение ключа

void setKey(const K &key)

{

\_key = key;

}

// Изменение данных

void setValue(const V &value)

{

\_value = value;

}

// Изменение следующего узла

void setNext(HashNode \*next)

{

\_next = next;

}

};

## Приложение 7 (класс HashTable)

// Константа размера хэш-таблицы

const size\_t SIZE = 100;

// Хэш-функция

struct KeyHash

{

unsigned long operator()(const std::string& key) const

{

int sum = 0;

for (char c : key)

sum += (int)c;

return sum % SIZE;

}

};

// Класс хэш-таблицы

template <typename K, typename V, typename F>

class HashTable

{

HashNode<K, V> \*\*table; // Таблица

F function; // Хэш-функция

public:

// Конструктор хэш-таблицы

HashTable()

{

table = new HashNode<K, V>\*[SIZE]();

}

bool existsKey(const K &key)

{

if (table[function(key)])

return true;

else

return false;

}

// Поиск по ключу key и запись в value

bool get(const K &key, V &value)

{

// Вычисление хэш-значения

unsigned long hashValue = function(key);

HashNode<K, V> \*entry = table[hashValue];

int count = 0;

while (entry)

{

count++;

if (entry->key() == key) // Если найдена запись

{

value = entry->value();

std::cout << "Хэш-таблица - операций поиска: " << count << std::endl;

return true;

}

entry = entry->next();

}

std::cout << "Полином с таким ключом не найден\n";

return false;

}

void add(const K &key, const V &value)

{

// Вычисление хэш-значения

unsigned long hashValue = function(key);

HashNode<K, V> \*prev = NULL;

HashNode<K, V> \*entry = table[hashValue];

// Вычисление количества операций

int count = 1;

while (entry && entry->key() != key)

{

prev = entry;

entry = entry->next();

count++;

}

// Изменение таблицы (перезапись или создание узла)

if (!entry)

{

entry = new HashNode<K, V>(key, value);

if (prev == NULL)

table[hashValue] = entry;

else

prev->setNext(entry);

}

else

entry->setValue(value);

std::cout << "Хэш-таблица - операций добавления: "

<< count << std::endl;

}

## };