**Московский Авиационный институт**
**(Национальный исследовательский университет)**

**Факультет №8**
**«Компьютерные науки и прикладная математика»**

**Кафедра 806**
**«Вычислительная математика и программирование»**

# Курсовая работа

по теме

«Разреженные матрицы»



| | |
|---|---|
| Студент: | Концебалов О.С. |
| Группа: | М8О-109Б-22 |
| Преподаватель: | Сысоев М. А. |
| Подпись: | |
| Оценка: | |



Москва, 2023

## Задание

Составить программу на языке Си с функциями для обработки прямоугольных разреженных матриц с элементами целого типа, которая:

Вводит матрицы различного размера с одновременным размещением ненулевых элементов в разреженной матрице в соответствии с заданной схемой;
Печатает введенные матрицы во внутреннем представлении и в обычном виде;
выполняет необходимые преобразования разреженных матриц (или вычисления над ними) путем обращения к соответствующим функциям;
Печатает результат преобразования во внутреннем представлении и в обычном виде.
В процедурах и функциях предусмотреть проверки и печать сообщений в случаях ошибок в задании параметров. Для отладки использовать матрицы, содержащие 5- 10% ненулевых элементов, с максимальным числом элементов 100.

## Вариант схемы размещения матрицы:

2. Один вектор:

Ненулевому элементу соответствуют две ячейки: первая содержит номер столбца, вторая содержит значение элемента. Нуль в первой ячейке означает конец строки, а вторая ячейка содержит в этом случае номер следующей хранимой строки. Нули в обеих ячейках являются признаком конца перечня ненулевых элементов разреженной матрицы.

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | Номер строки | Номер столбца | Значение | Номер столбца | Значение | ... |

...

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | Номер строки | Номер столбца | Значение | ... | 0 | 0 |

## Вариант преобразования:

5. Умножить вектор-строку на разреженную матрицу и вычислить количество ненулевых элементов результата

## Общий метод решения

Разреженная матрица — это матрица с преимущественно нулевыми элементами. В противном случае, если большая часть элементов матрицы ненулевые, матрица считается плотной.
Хранение разреженной матрицы в памяти должно обеспечивать:

1. экономию памяти
2. быстрый доступ к нулевым и ненулевым элементам по их индексу.

Поэтому, хранение разреженной матрицы с помощью одного вектора крайне удобно. Мы представляем исходную матрицу M размеров m x n, содержащую некоторое число ненулевых значений в виде одного вектора. В нем каждому

элементу соответствуют две ячейки – первая это номер столбца, а вторая значение элемента. Если в первой ячейке ноль, то строка закончилась, а во второй будет номер следующей строки. Нули в обеих ячейках означают, что ненулевые элементы матрицы кончились.

## Структура:

1. Класс Vector, похожий на std::vector из стандартной библиотеки, хранящий шаблонный тип.
2. Класс Matrix, предназначенный для работы с разреженными матрицами.
3. main - программа для интерактивной работы.
4. benchmark - тест-сравнение времени работы стандартного и пользовательского вектора.

## Алгоритм решения поставленной задачи

Для обработки разреженных матриц опишем класс вектора с его множеством операций и реализуем вектор на C++. Отдельно опишем класс для обработки разреженных матриц:

1. Считывание матриц в обычном виде из файла с преобразованием в векторы согласно заданной схеме размещения.
2. Печать матрицы в естественном виде.
3. Печать вектора (схема размещения ненулевых элементов разреженной матрицы).
4. Выполнение заданного преобразования

## Код программы

**myVector.hpp**

```
#ifndef MYVECTORHPP
#define MYVECTORHPP
#include <iostream>

template <typename T>
class myVector{
private:
    T* data;
    size_t vec_size;
    size_t vec_capacity;

public:
    myVector();
    myVector(const myVector<T>& another_vector);
    myVector(const std::initializer_list<T>& list);
```

```
    ~myVector();

    void reserve(size_t n);
    void resize(size_t new_size, const T& value = T());
    void shrink_to_fit();
    void push_back(const T& value);
    void pop_back();
    T& at(size_t index);
    const T& front() const;
    const T& back() const;
    bool empty() const;
    size_t size() const;
    size_t capacity() const;
    void clear();

    template <typename... Args>
    void emplace_back(const Args& ...args);

    T& operator[](size_t i) const;
    bool operator==(const myVector<T>& another_vector);
    bool operator!=(const myVector<T>& another_vector);
    void erase(size_t element_index);
    void erase(size_t start_index, size_t end_index);
    void swap(myVector<T>& another_vector);
};

#endif
```

**myVector.cpp**

```
#include "MyVector.hpp"
#include <climits>
#include <memory>

template <typename T>
myVector<T>::myVector(){
    data = reinterpret_cast<T*>(new int8_t[sizeof(T)]);
    vec_size = 0;
    vec_capacity = 1;
}
```

```
template <typename T>
myVector<T>::myVector(const myVector<T>& another_vector){
    resize(another_vector.size());

    for (size_t i = 0; i != vec_size; ++i){
        data[i].~T();
        new (data + i) T(another_vector[i]);
    }
}

template <typename T>
myVector<T>::myVector(const std::initializer_list<T>& list){
    T* new_data = reinterpret_cast<T*>(new int8_t[list.size() *
sizeof(T)]);

    try{
        std::uninitialized_copy(list.begin(), list.end(),
new_data);
    } catch (...){
        delete[] reinterpret_cast<int8_t*>(new_data);
        throw;
    }

    delete[] reinterpret_cast<int8_t*>(data);
    data = new_data;
    vec_size = list.size();
    vec_capacity = list.size();
}

template <typename T>
myVector<T>::~myVector(){
    for (size_t i = 0; i != vec_size; ++i){
        data[i].~T();
    }

    delete[] reinterpret_cast<int8_t*>(data);
}

template <typename T>
void myVector<T>::reserve(size_t reserved_size){
```

```
    if (reserved_size > ULLONG_MAX){
        throw std::range_error("Capacity overflow");
    }

    if (reserved_size <= vec_capacity) return;

    T* new_data = reinterpret_cast<T*>(new int8_t[reserved_size
* sizeof(T)]);
    try{
        std::uninitialized_copy(data, data + vec_size,
new_data);
    } catch(...){
        delete[] reinterpret_cast<int8_t*>(new_data);
        throw;
    }

    for (size_t i = 0; i != vec_size; ++i){
        data[i].~T();
    }

    delete[] reinterpret_cast<int8_t*>(data);
    data = new_data;
    vec_capacity = reserved_size;
}

template <typename T>
void myVector<T>::resize(size_t new_size, const T& value){
    if (new_size > ULLONG_MAX){
        throw std::range_error("Capacity overflow");
    }

    if (new_size > vec_capacity) reserve(new_size);

    if (new_size > vec_size){
        for (size_t i = vec_size; i != new_size; ++i){
            new (data + i) T(value);
        }
    }

    if (new_size < vec_size){
```

```
        for (size_t i = new_size; i != vec_size; ++i){
            data[i].~T();
        }
    }

    vec_size = new_size;
}

template <typename T>
void myVector<T>::shrink_to_fit(){
    if (vec_size < vec_capacity){
        T* new_data = reinterpret_cast<T*>(new int8_t[vec_size
* sizeof(T)]);

        for (size_t i = 0; i != vec_size; ++i){
            new (new_data + i) T(data[i]);
        }

        delete[] reinterpret_cast<int8_t*>(data);
        data = new_data;
        vec_capacity = vec_size;
    }
}

template <typename T>
void myVector<T>::push_back(const T& value){
    if (vec_capacity == vec_size){
        if (vec_capacity * 2 < vec_capacity){
            throw std::range_error("Capacity overflow");
        }

        reserve(2 * vec_capacity);
    }

    new (data + vec_size) T(value);
    ++vec_size;
}

template <typename T>
void myVector<T>::pop_back(){
```

```
    if (vec_size == 0) return;

    --vec_size;
    data[vec_size].~T();
}

template <typename T>
T& myVector<T>::at(size_t index){
    if (index >= vec_size){
        throw std::out_of_range("Index out of range");
    }

    return data[index];
}

template <typename T>
const T& myVector<T>::front() const{
    if (vec_size == 0){
        throw std::range_error("Vector is empty");
    }

    return data[0];
}

template <typename T>
const T& myVector<T>::back() const{
    if (vec_size == 0){
        throw std::range_error("Vector is empty");
    }

    return data[vec_size - 1];
}

template <typename T>
bool myVector<T>::empty() const{
    return vec_size == 0;
}

template <typename T>
size_t myVector<T>::size() const{
```

```
    return vec_size;
}

template <typename T>
size_t myVector<T>::capacity() const{
    return vec_capacity;
}

template <typename T>
void myVector<T>::clear(){
    for (size_t i = 0; i != vec_size; ++i){
        data[i].~T();
    }
    vec_size = 0;
}

template <typename T>
template <typename... Args>
void myVector<T>::emplace_back(const Args& ...args){
    if (vec_capacity == vec_size) reserve(2 * vec_capacity);

    new (data + vec_size) T(args...);
    ++vec_size;
}

template <typename T>
T& myVector<T>::operator[](size_t i) const{
    return data[i];
}

template <typename T>
bool myVector<T>::operator==(const myVector<T>&
another_vector){
    if (vec_size != another_vector.vec_size) return false;

    for (size_t i = 0; i != vec_size; ++i){
        if (this->data[i] != another_vector.data[i]) return
false;
    }
```

```
    return true;
}

template <typename T>
bool myVector<T>::operator!=(const myVector<T>&
another_vector){
    if (vec_size != another_vector.vec_size) return true;

    for (size_t i = 0; i != vec_size; ++i){
        if (this->data[i] != another_vector.data[i]) return
true;
    }

    return false;
}

template <typename T>
void myVector<T>::erase(size_t element_index){
    if (element_index >= vec_size){
        throw std::out_of_range("Index out of range");
    }

    for (size_t i = element_index; i != vec_size - 1; ++i){
        data[i].~T();
        data[i] = data[i + 1];
    }
    --vec_size;
    resize(vec_size);
}

template <typename T>
void myVector<T>::erase(size_t start, size_t end){
    if (start > end){
        throw std::invalid_argument("Start index must be less
than end index");
    }

    if (start > vec_size || end > vec_size){
        throw std::out_of_range("Index out of range");
    }
```

```
    size_t to_delet = end - start + 1;
    for (size_t i = start; i != vec_size - to_delet; ++i){
        data[i].~T();
        data[i] = data[i + to_delet];
    }
    vec_size -= to_delet;
    resize(vec_size);
}

template <typename T>
void myVector<T>::swap(myVector<T>& another_vector){
    T* temp_data = data;
    data = another_vector.data;
    another_vector.data = temp_data;

    size_t temp_size = vec_size;
    vec_size = another_vector.vec_size;
    another_vector.vec_size = temp_size;

    size_t temp_capacity = vec_capacity;
    vec_capacity = another_vector.vec_capacity;
    another_vector.vec_capacity = temp_capacity;
}
```

**Matrix.hpp**

```
#ifndef MATRIXHPP
#define MATRIXHPP
#include "MyVector.hpp"
#include <iostream>

class Matrix{
private:
    myVector<int> line_matrix;
    size_t matrix_lines;
    size_t matrix_columns;

public:
    Matrix();
    Matrix(size_t lines, size_t columns);
```

```
    friend std::istream& operator>>(std::istream& is, Matrix&
matrix);
    friend std::ostream& operator<<(std::ostream& os, Matrix&
matrix);

    myVector<int> multiply(const myVector<int>& vector_string);
    void print();
};

#endif
```

**Matrix.cpp**

```
#include "Matrix.hpp"
#include <iostream>

Matrix::Matrix(): matrix_lines(0), matrix_columns(0){
    line_matrix.push_back(0);
    line_matrix.push_back(0);
}

Matrix::Matrix(size_t lines, size_t columns):
matrix_lines(lines), matrix_columns(columns){
    line_matrix.push_back(0);
    line_matrix.push_back(0);
}

std::istream& operator>>(std::istream& is, Matrix& matrix){
    matrix.line_matrix.pop_back();
    size_t line, columns;
    int input_value;

    is >> line >> columns;

    matrix.matrix_lines = line;
    matrix.matrix_columns = columns;

    for (size_t i = 0; i != line; ++i){
        for (size_t j = 0; j != columns; ++j){
            is >> input_value;
```

```
            if (input_value != 0){
                if (matrix.line_matrix.back() == 0){
                    matrix.line_matrix.push_back(i + 1);
                }
                matrix.line_matrix.push_back(j + 1);
                matrix.line_matrix.push_back(input_value);
            }
        }

        if (matrix.line_matrix.back() != 0){
            matrix.line_matrix.push_back(0);
        }
    }
    matrix.line_matrix.push_back(0);

    return is;
}

std::ostream& operator<<(std::ostream& os, Matrix& matrix){
    for (size_t i = 0; i != matrix.line_matrix.size(); ++i){
        os << matrix.line_matrix[i] << " ";
    }

    return os;
}

myVector<int> Matrix::multiply(const myVector<int>&
vector_string){
    if (vector_string.size() != matrix_lines){
        throw std::range_error("Vector columns not equal matrix
lines");
    }

    myVector<int> result;

    for (size_t i = 0; i != matrix_columns; ++i){
        result.push_back(0);
    }

    size_t cur_line = 0;
```

```
    while (cur_line < line_matrix.size()){
        int j = line_matrix[cur_line];
        if (j == 0){
            ++cur_line;
            if (cur_line >= line_matrix.size() ||
line_matrix[cur_line] == 0) break;
            j = line_matrix[cur_line];
        }

        int cur_column = cur_line + 1;
        int column_of_result_vector = line_matrix[cur_line] -
1;

        for (size_t column = cur_column; line_matrix[column] !=
0; column += 2){
            int value = line_matrix[column + 1];
            result[line_matrix[column] - 1] +=
(vector_string[column_of_result_vector] * value);
            cur_line = column;
        }

        ++cur_line;
    }

    uint64_t number_of_non_zero_elements = 0;
    std::cout << "Result: ( ";
    for (size_t i = 0; i != result.size(); ++i){
        if (result[i] != 0) ++number_of_non_zero_elements;
        std::cout << result[i] << " ";
    }
    std::cout << ")\nNumber of non-zeros elements of the
result: " << number_of_non_zero_elements << "\n";

    return result;
}

void Matrix::print(){
    size_t line = 1, column = 1;
    for (size_t i = 0; i < line_matrix.size(); ++i) {
```

```
        if (!line_matrix[i]) {
            ++i;
            if (!line_matrix[i]) {
                while (line <= matrix_lines) {
                    while (column <= matrix_columns) {
                        column++;
                        std::cout << 0 << " ";
                    }
                    std::cout << '\n';
                    line++;
                    column = 1;
                }
                return;
            }
            while (line < line_matrix[i]) {
                while (column <= matrix_columns) {
                    column++;
                    std::cout << 0 << " ";
                }
                std::cout << '\n';
                line++;
                column = 1;
            }
        }
        else {
            while (column != line_matrix[i]) {
                column++;
                std::cout << 0 << " ";
            }
            ++i;
            column++;
            std::cout << line_matrix[i] << " ";
        }
    }
}
```

**Benchmark.cpp**

```
#include "MyVector.hpp"
#include <algorithm>
```

```cpp
#include <chrono>
#include <fstream>
#include <iostream>
#include <vector>

void benchmark(){
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::cout.tie(nullptr);
    std::ifstream tests_file("..\\test.txt", std::ios::in);

    if (!tests_file.is_open()){
        std::cerr << "ERROR OPENING FILE\n";
        return;
    }

    myVector<int> my_vector;
    std::vector<int> original_vector;
    int amount_of_numbers, number;

    std::cout << "\nComparing \"myVector\" and the \"vector\"
from STL\n\n";
    tests_file >> amount_of_numbers;
    std::cout << "Amount of numbers: " << amount_of_numbers <<
"\n\n";

    std::cout << "push_back, complexity O(1)\n";
    std::chrono::steady_clock::time_point start_time =
std::chrono::steady_clock::now();
    for (int i = 0; i != amount_of_numbers; ++i){
        tests_file >> number;
        my_vector.push_back(number);
    }
    std::chrono::steady_clock::time_point end_time =
std::chrono::steady_clock::now();
    std::cout << "Time of work my_vector: " <<
        std::chrono::duration_cast<std::chrono::milliseconds>(e
nd_time - start_time).count() << " milliseconds\n";

    tests_file.clear();
```

```
    tests_file.seekg(0);
    tests_file >> number;
    start_time = std::chrono::steady_clock::now();
    for (int i = 0; i != amount_of_numbers; ++i){
        tests_file >> number;
        original_vector.push_back(number);
    }
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of work original_vector: " <<
        std::chrono::duration_cast<std::chrono::milliseconds>(e
nd_time - start_time).count() << " milliseconds\n";

    if (my_vector.size() != original_vector.size()){
        throw std::exception();
    }

    for (size_t i = 0; i != original_vector.size(); ++i){
        if (my_vector[i] != original_vector[i]){
            std::cout << "ERROR: index " << i << " my_vector[i]
= " << my_vector[i] <<
                    ", original_vector[i] = " <<
original_vector[i] << "\n";
            throw std::exception();
        }
    }

    std::cout << "pop_back, complexity O(1)\n";
    start_time = std::chrono::steady_clock::now();
    for (int i = 0; i != amount_of_numbers; ++i){
        my_vector.pop_back();
    }
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of work my_vector: " <<
        std::chrono::duration_cast<std::chrono::milliseconds>(e
nd_time - start_time).count() << " milliseconds\n";

    start_time = std::chrono::steady_clock::now();
    for (int i = 0; i != amount_of_numbers; ++i){
        original_vector.pop_back();
    }
```

```
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of work original_vector: " <<
        std::chrono::duration_cast<std::chrono::milliseconds>(e
nd_time - start_time).count() << " milliseconds\n";

    if (my_vector.size() != original_vector.size()){
        throw std::exception();
    }

    for (size_t i = 0; i != original_vector.size(); ++i){
        if (my_vector[i] != original_vector[i]){
            std::cout << "ERROR: index " << i << " my_vector[i]
= " << my_vector[i] <<
                    ", original_vector[i] = " <<
original_vector[i] << "\n";
            throw std::exception();
        }
    }

    std::cout << "shrink_to_fit\n";
    start_time = std::chrono::steady_clock::now();
    for (int i = 0; i != amount_of_numbers; ++i){
        my_vector.shrink_to_fit();
    }
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of work my_vector: " <<
        std::chrono::duration_cast<std::chrono::milliseconds>(e
nd_time - start_time).count() << " milliseconds\n";

    start_time = std::chrono::steady_clock::now();
    for (int i = 0; i != amount_of_numbers; ++i){
        original_vector.shrink_to_fit();
    }
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of work original_vector: " <<
        std::chrono::duration_cast<std::chrono::milliseconds>(e
nd_time - start_time).count() << " milliseconds\n";

    if (my_vector.capacity() != original_vector.capacity()){
```

```
        std::cout << "ERROR: my_vector.capacity() = " <<
my_vector.capacity() <<
                ", original_vector.capacity() = " <<
original_vector.capacity() << "\n";
    }

    std::cout << "reserve\n";
    start_time = std::chrono::steady_clock::now();
    my_vector.reserve(2 * amount_of_numbers);
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of work my_vector: " <<
        std::chrono::duration_cast<std::chrono::milliseconds>(e
nd_time - start_time).count() << " milliseconds\n";

    start_time = std::chrono::steady_clock::now();
    original_vector.reserve(2 * amount_of_numbers);
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of work original_vector: " <<
        std::chrono::duration_cast<std::chrono::milliseconds>(e
nd_time - start_time).count() << " milliseconds\n";
    std::cout << "my_vector.capacity() = " <<
my_vector.capacity() <<
        ", original_vector.capacity() = " <<
original_vector.capacity() << "\n";

    std::cout << "resize\n";
    start_time = std::chrono::steady_clock::now();
    my_vector.resize(2 * amount_of_numbers, number);
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of work my_vector: " <<
        std::chrono::duration_cast<std::chrono::milliseconds>(e
nd_time - start_time).count() << " milliseconds\n";

    start_time = std::chrono::steady_clock::now();
    original_vector.resize(2 * amount_of_numbers, number);
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of work original_vector: " <<
        std::chrono::duration_cast<std::chrono::milliseconds>(e
nd_time - start_time).count() << " milliseconds\n";
```

```
    if (my_vector.size() != original_vector.size()){
        throw std::exception();
    }

    for (size_t i = 0; i != original_vector.size(); ++i){
        if (my_vector[i] != original_vector[i]){
            std::cout << "ERROR: index " << i << " my_vector[i]
= " << my_vector[i] <<
                ", original_vector[i] = " << original_vector[i]
<< "\n";
            throw std::exception();
        }
    }
    std::cout << "my_vector.capacity() = " <<
my_vector.capacity() <<
        ", original_vector.capacity() = " <<
original_vector.capacity() << "\n";
    std::cout << "my_vector.size() = " << my_vector.size() <<
        ", original_vector.size() = " << original_vector.size()
<< "\n";

    std::cout << "clear\n";
    start_time = std::chrono::steady_clock::now();
    my_vector.clear();
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of work my_vector: " <<
        std::chrono::duration_cast<std::chrono::milliseconds>(e
nd_time - start_time).count() << " milliseconds\n";

    start_time = std::chrono::steady_clock::now();
    original_vector.clear();
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of work original_vector: " <<
        std::chrono::duration_cast<std::chrono::milliseconds>(e
nd_time - start_time).count() << " milliseconds\n";

    if (my_vector.size() != 0 && original_vector.size() != 0){
        throw std::exception();
    }
```

```
    std::cout << "my_vector.capacity() = " <<
my_vector.capacity() <<
        ", original_vector.capacity() = " <<
original_vector.capacity() << "\n";
    std::cout << "my_vector.size() = " << my_vector.size() <<
        ", original_vector.size() = " << original_vector.size()
<< "\n";

    std::cout << "copy constructor\n";
    start_time = std::chrono::steady_clock::now();
    myVector<int> my_vector1(my_vector);
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of work my_vector: " <<
        std::chrono::duration_cast<std::chrono::milliseconds>(e
nd_time - start_time).count() << " milliseconds\n";

    start_time = std::chrono::steady_clock::now();
    std::vector<int> original_vector1(original_vector);
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of work original_vector: " <<
        std::chrono::duration_cast<std::chrono::milliseconds>(e
nd_time - start_time).count() << " milliseconds\n";

    std::cout << "equality\n";
    start_time = std::chrono::steady_clock::now();
    std::cout << (my_vector == my_vector1) << "\n";
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of work my_vector: " <<
        std::chrono::duration_cast<std::chrono::milliseconds>(e
nd_time - start_time).count() << " milliseconds\n";

    start_time = std::chrono::steady_clock::now();
    std::cout << (original_vector == original_vector1) << "\n";
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of work original_vector: " <<
        std::chrono::duration_cast<std::chrono::milliseconds>(e
nd_time - start_time).count() << " milliseconds\n";

    std::cout << "swap\n";
    start_time = std::chrono::steady_clock::now();
```

```
    my_vector.swap(my_vector1);
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of work my_vector: " <<
        std::chrono::duration_cast<std::chrono::milliseconds>(e
nd_time - start_time).count() << " milliseconds\n";

    start_time = std::chrono::steady_clock::now();
    original_vector.swap(original_vector1);
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of work original_vector: " <<
        std::chrono::duration_cast<std::chrono::milliseconds>(e
nd_time - start_time).count() << " milliseconds\n";

    return;
}
```

**Run.cpp**

```
#include "MyVector.hpp"
#include "MyVector.cpp"
#include "benchmark.cpp"
#include "Matrix.hpp"
#include "Matrix.cpp"
#include <iostream>

int main(){

    myVector<int> vector_string = {4, 10, 52, 0, 34, 47, -4,
9};
    Matrix sparse_matrix;

    std::cin >> sparse_matrix;
    myVector<int> res = sparse_matrix.multiply(vector_string);

    benchmark();
    return 0;
}
```

## Тестирование

**1) vector_string = {4, 10, 52, 0, 34, 47, -4, 9};**

```
8 10
0 0 0 0 66 8 66 0 0 95
46 0 4 0 71 4 0 0 63 0
23 6 64 1 23 0 0 0 0 60
34 0 0 0 78 0 0 0 20 65
0 4 98 0 0 72 0 0 42 0
0 44 0 2 4 96 0 0 32 0
0 63 0 0 0 87 71 91 0 0
18 0 9 0 61 0 73 94 0 0
Result: ( 1818 2264 6781 146 2907 6684 637 482 3562 3500 )
Number of non-zeros elements of the result: 10
```

**2) vector_string = {0, 34, 47, -4, 9};**

```
5 5
0 97 0 46 0
48 0 24 0 91
0 0 22 92 0
48 0 57 42 0
0 45 0 53 0
Result: ( 1440 405 1622 4633 3094 )
Number of non-zeros elements of the result: 5
```

## Тесты производительности

Количество чисел для сравнения: 1751438

**push_back, complexity O(1)**
Time of work my_vector: 695 milliseconds
Time of work original_vector: 751 milliseconds

**pop_back, complexity O(1)**
Time of work my_vector: 2 milliseconds
Time of work original_vector: 6 milliseconds

**shrink_to_fit**
Time of work my_vector: 4 milliseconds
Time of work original_vector: 10 milliseconds

**reserve**
Time of work my_vector: 0 milliseconds
Time of work original_vector: 0 milliseconds
my_vector.capacity() = 3502876
original_vector.capacity() = 3502876

**resize**
Time of work my_vector: 8 milliseconds
Time of work original_vector: 3 milliseconds
my_vector.capacity() = 3502876
original_vector.capacity() = 3502876
my_vector.size() = 3502876
original_vector.size() = 3502876

**clear**
Time of work my_vector: 2 milliseconds
Time of work original_vector: 0 milliseconds
my_vector.capacity() = 3502876
original_vector.capacity() = 3502876
my_vector.size() = 0
original_vector.size() = 0

**copy constructor**
Time of work my_vector: 0 milliseconds
Time of work original_vector: 0 milliseconds

**equality**
Time of work my_vector: 0 milliseconds
Time of work original_vector: 0 milliseconds

**swap**
Time of work my_vector: 0 milliseconds
Time of work original_vector: 0 milliseconds

Как мы видим из результатов тестирования, мой кастомный вектор работает быстрее вектора из STL примерно в 1,5-2 раза (удивился). Думаю это связано с тем, что наши реализации очень похоже друг на друга, но вектор из STL будет использовать более безопасные и оптимальные функции и методы, чем мой. Также вектор из STL написан полностью на аллокаторах и использует мув семантику, а я их в своей реализации не использовал, может быть это тоже как-то влияет.

Значительные отличия по скорости отличаются методы push_back и pop_back, а также shrink_to_fit и clear. Shrink_to_fit осуществляет вызов деструктора объектов в цикле, что может занимать значительное время при большом количестве элементов, в стандартном векторе реализация метода скорее всего неким образом более оптимизирована. В pop_back я просто декрементирую size и вызываю деструктор элемента по индексу size, видимо в STL векторе это реализовано как-то иначе. В push_back я делаю placement new нового элемента, предварительно проверив capacity и size. Предположительно мой работает быстрее, потому что различается время работы reserve стандартного вектора и моего.

Еще стоит отметить различия во времени работы resize. Мой работает медленнее возможно потому что я вызываю деструкторы от объектов в цикле, что может занимать довольно много времени.

## Заключение

В результате получилось выполнить поставленное задание — реализовал оптимальное хранение разреженной матрицы и написал функцию для выполнения умножения вектора-строки на матрицу.

В ходе работы написал свой вектор, который оказался даже побыстрее чем вектор из STL (шок), что было очень интересно. Благодаря этому я получил хорошие навыки работы с динамическими структурами и опыт работы с памятью, которые очень пригодятся мне в дальнейшем.