|  |  |
| --- | --- |
| **Gerb-BMSTU_01** | **Министерство образования и науки Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ Информатика и системы управления (ИУ)

КАФЕДРА Программное обеспечение ЭВМ и информационные технологии (ИУ7)

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №3  
«Обработка разреженных матриц»**

Студент, группа **Пронин А.С. ИУ7-32Б**

*2020 г.*

# Условия задачи

Разреженная (содержащая много нулей) матрица хранится в форме 3-х объектов:

• вектор A содержит значения ненулевых элементов;

• вектор IA содержит номера строк для элементов вектора A;

•связный список JA, в элементе Nk которого находится номер компонент в A и IA, с которых начинается описание столбца Nk матрицы A.

1. Смоделировать операцию сложения двух матриц, хранящихся в этой форме, с получением результата в той же форме.

2. Произвести операцию сложения, применяя стандартный алгоритм работы с матрицами.

3. Сравнить время выполнения операций и объем памяти при использовании этих 2-х алгоритмов при различном проценте заполнения матриц.

# Техническое задание

## Входные данные

1) Целое число от 0 до 7 (для ввода номера команды из меню)

2) Целое число от 1 до 5000 (для ввода кол-ва строк и столбцов)

3) Целое число от 1 до 100 (для ввода процента разреженности)

4) Целое число от 1 до максимального кол-ва элементов в матрице (rows\*columns) (для ввода кол-ва ненулевых элементов разреженной матрицы)

5) Целые числа через пробел (для ввода элементов разреженной матрицы)

6) Целые числа от 0 до кол-ва строк введённых ранее через пробел (для ввода номеров строк для элементов разреженной матрицы)

6) Целые числа через пробел (для ввода элементов связного списка JA)

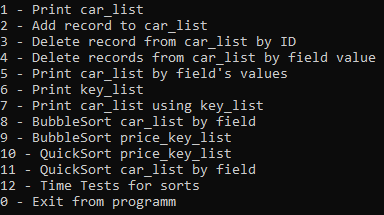
## Выходные данные

Вывод матриц в обычном или разреженном виде

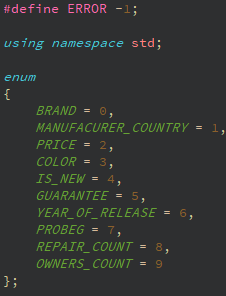
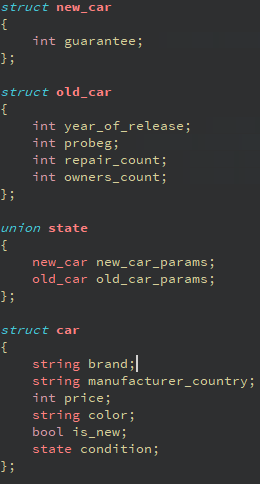
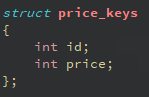
Время выполнения сложения матриц обычным способом и методом разреженных матриц

## Функции программы

Программа представляет из себя консольное приложение исполняющее следующие команды:



## Структуры данных

# Описание алгоритма

Вся информация о таблице записей о машинах хранится в динамическом массиве типа car\* (структура car описана выше). Также информация о таблице ключей соответственно в динамическом массиве типа price\_keys\* (тоже описанного выше). А далее описаны функции, работающие с этими массивами.

### Набор функций, используемых в меню:

1. void **print\_car\_list**(car \*cars\_list, int size\_of\_list)

*Функция* *вывода* *текущего* *списка* *машин*

1. int **read\_record** (string input, car \*car\_record\_rez)

*Функция* *считывания* *записи* *о* *машине* *из* *строки*

1. int **add\_record** (car \*\*car\_list, int size\_of\_list, car car\_record)

*Функция* *добавления* *одной* *записи* *в* *конец* *списка* *машин*

1. int **delete\_record**(car \*\*car\_list, int size\_of\_list, int id)

*Функция* *удаления* *записи* *из* *списка* *машин* *по* *номеру*

1. int **delete\_car\_record\_by\_pole**(int pole, string znach, car \*\*car\_list, int \*size\_of\_list) *Функция* *удаления* *записи* *из* *списка* *машин* *по* *значению* *конкретного* *поля (использует* **delete\_record**()*)*
2. int **print\_car\_record\_by\_poles**(int pole[10], string znach[10], car \*\*car\_list, int size\_of\_list) *Функция* *вывода* *списка* *машин* *по* *значениям* *одного* *или* *нескольких* *полей*
3. void **print\_price\_key\_list**(price\_keys \*key\_list, int size\_of\_list)

*Функция* *вывода* *таблицы* *ключей*

1. void **print\_car\_list\_by\_price\_keys**(car \*cars\_list, int size\_of\_list, price\_keys \*key\_list) *Функция* *вывода* *списка* *машин* *с* *использованием* *таблицы* *ключей*
2. void **car\_sort\_puz**(car \*\*mas, int size\_of\_mas, int (\*comparator)(*const* car\*, *const* car\*)) *Функция* *сортировки* *списка* *машин* *пузырьком*
3. void **price\_keys\_sort\_puz**(price\_keys \*\*mas, int size\_of\_mas)

*Функция* *сортировки* *таблицы* *ключей* *пузырьком*

1. void **price\_keys\_sort\_qsort**(price\_keys \*a, int l, int r)

*Функция* *сортировки* *таблицы* *ключей* *быстрой* *сортировкой*

1. void **car\_sort\_qsort**(car \*a, int l, int r, int (\*comparator)(*const* car\*, *const* car\*)) *Функция* *сортировки* *списка* *машин* *быстрой* *сортировкой*
2. int **read\_file**(string file\_name, car \*\*car\_list, int \*size\_of\_list) *Функция* *считывания* *списка* *машин* *из* *файла (Использует* **read\_record**() и **add\_record**()*)*
3. void **car\_sort\_puz\_test**(car \*\*mas, int size\_of\_mas, int (\*comparator)(*const* car\*, *const* car\*), int mn) *Функция* *тестирования* *сортировки* *списка* *машин* *пузырьком* *на* *время*
4. void **price\_keys\_sort\_puz\_test**(price\_keys \*\*mas, int size\_of\_mas, int mn) *Функция* *тестирования* *сортировки* *таблицы* *ключей* *пузырьком* *на* *время*
5. void **car\_sort\_qsort\_test**(car \*a, int l, int r, int (\*comparator)(*const* car\*, *const* car\*), int mn) *Функция* *тестирования* *сортировки* *списка* *машин* *быстрой* *сортировкой* *на* *время*
6. void **price\_keys\_sort\_qsort\_test**(price\_keys \*a, int l, int r, int mn) *Функция* *тестирования* *сортировки* *таблицы* *ключей* *быстрой* *сортировкой* *на* *время*

### Вспомогательные функции:

1. void **print\_car\_record**(car car\_record)

*Функция* *вывода* *отдельной* *записи* *о* *машине*

1. int **string\_to\_int**(string input, int \*rez)

*Функция* *перевода* *строки* *в* *число*

1. void **car\_list\_copy** (car \*car\_list1, car \*car\_list2, int size\_of\_list)

*Функция* *копирования* *списка* *машин*

1. void **swap\_car\_records**(car \*car\_list1, car \*car\_list2)

*Функция* *обмена* *записей* *о* *машинах* *местами*

1. void **swap\_price\_keys**(price\_keys \*key1, price\_keys \*key2)

*Функция* *обмена* *ключей* *местами* *в* *таблице*

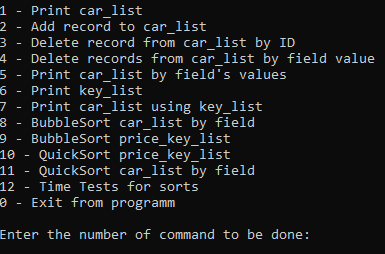
### Функции сравнения:

1. int **comp\_string**(*const* string \*str1, *const* string \*str2)
2. int **comp\_car\_brand**(*const* car \*i, *const* car \*j)
3. int **comp\_car\_manufacturer\_country**(*const* car \*i, *const* car \*j)
4. int **comp\_car\_price**(*const* car \*i, *const* car \*j)
5. int **comp\_car\_color**(*const* car \*i, *const* car \*j)
6. int **comp\_car\_is\_new**(*const* car \*i, *const* car \*j)
7. int **comp\_car\_guarantee**(*const* car \*i, *const* car \*j)
8. int **comp\_car\_year\_of\_release**(*const* car \*i, *const* car \*j)
9. int **comp\_car\_probeg**(*const* car \*i, *const* car \*j)
10. int **comp\_car\_repair\_count**(*const* car \*i, *const* car \*j)
11. int **comp\_car\_owners\_count**(*const* car \*i, *const* car \*j)

*Описание функции вытекает из её названия*

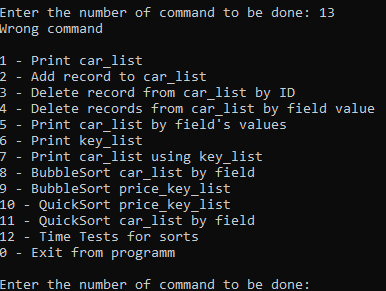
# Тесты

### Вот так выглядит меню:



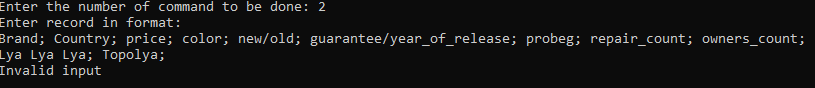
### Некорректный ввод:

1. Некорректная команда





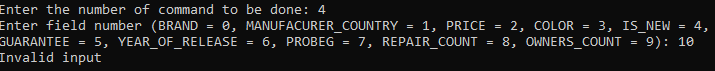
1. Команда 2 (добавление записи)

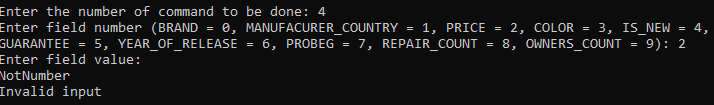


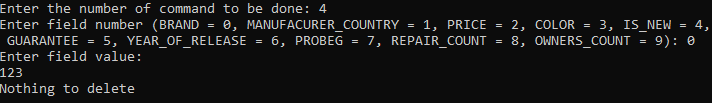
1. Команда 3 (Удаление записи по номеру)



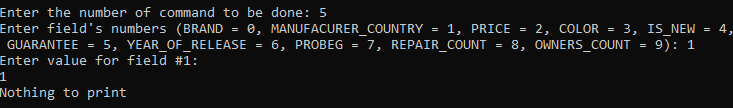
1. Команда 4 (Удаление записи по значению поля)

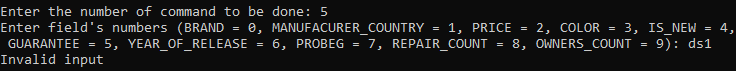




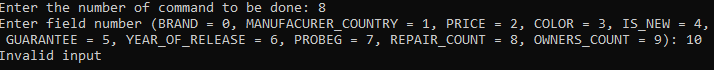


1. Команда 5 (вывод списка машин по значениям полей)

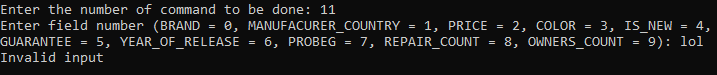




1. Команда 8 (сортировка списка машин пузырьком)



1. Команда 11 (сортировка списка машин быстрой сортировкой)

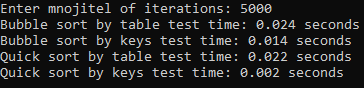


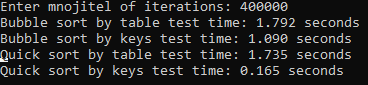
# Оценка эффективности

С помощью sizeof определим память занимаемую переменными типа **car** и **price\_keys**.

|  |  |  |
| --- | --- | --- |
|  | Список машин | Таблица ключей |
| Память (N – кол-во элементов) | N\*128 байт | N\*8 байт |

Из таблицы памяти мы видим, что таблица ключей увеличивает кол-во используемой памяти на ~6% (8 байт за каждую запись).





|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кол-во зап. | Список Bubble | Ключи Bubble | Список Qsort | Ключи Qsort |
| 5000\*40 | 0.024 sec | 0.014 sec | 0.022 sec | 0.002 sec |
| 400000\*40 | 1.792 sec | 1.090 sec | 1.735 sec | 0.165 sec |

* Список Bubble – void **car\_sort\_puz**(…)
* Ключи Bubble – void **price\_keys\_sort\_puz**(…)
* Список Qsort – void **car\_sort\_qsort**(…)
* Ключи Qsort – void **price\_keys\_sort\_qsort**(…)

Из таблицы времени выполнения сортировок мы видим, что при не очень большом списке эффективность таблицы ключей на ~71% (в 1.71 раз) больше при использовании BubbleSort, и на 1000% (в 11 раз) больше при использовании QuickSort. При значительном увеличении списка (в 80 раз больше чем в первом примере) эффективность таблицы ключей возрастает на ~64% (в 1.64 раз) при использовании BubbleSort, и на ~950% (в 10.5 раз) при использовании QuickSort. (Очевидно, что при большем кол-ве записей точность вычисления эффективности выше). Также можем сравнить эффективность сортировок по отдельности:

1. При не большом списке:

* QuickSort эффективней BubbleSort на 9% (в 1.09 раз) при использовании самого списка
* QuickSort эффективней BubbleSort на 600% (в 7 раз) при использовании таблицы ключей

1. При большом списке:

* QuickSort эффективней BubbleSort на 3% (в 1.03 раз) при использовании самого списка
* QuickSort эффективней BubbleSort на 560% (в 6.6 раз) при использовании самого списка

# Контрольные вопросы

**1.** **Что такое разреженная матрица, какие схемы хранения таких матриц Вы знаете?**

Разреженная матрица – это матрица, содержащая большое количество нулей. Способы хранения: связная схема хранения, строчный формат, столбцовый формат, линейный связный список, кольцевой связный список, двунаправленные стеки и очереди.

**2.** **Каким образом и сколько памяти выделяется под хранение разреженной и обычной матрицы?**

Под обычную матрицу выделяет N \* M ячеек памяти, где N – строки, а M – столбцы. Для разреженной матрицы – зависит от способа. В моём случае требуется 2 \* K + M ячеек памяти, где K – количество ненулевых элементов.

**3.** **Каков принцип обработки разреженной матрицы?**

Алгоритмы обработки разреженных матриц предусматривают действие только с ненулевыми элементами, и, таким образом, количество операций будет пропорционально количеству ненулевых элементов

**4.** **В каком случае для матриц эффективнее применять стандартные алгоритмы обработки матриц? От чего это зависит?**

Стандартные алгоритмы обработки матриц эффективнее применять при большом количестве ненулевых элементов. Cтоит отметить, что если не так важна память, занимаемая матрицами, но важно время, то в случае сложения лучше так же воспользоваться стандартными алгоритмами сложения матриц.

# Вывод