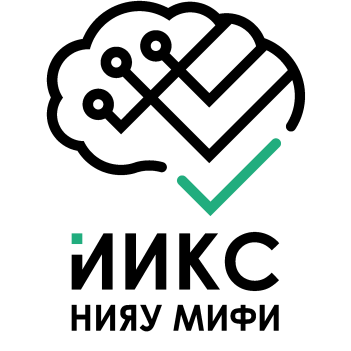
**Национальный исследовательский ядерный университет «МИФИ»**

**Институт интеллектуальных кибернетических систем**

**Кафедра №12 «Компьютерные системы и технологии»**



**ОТЧЕТ**

**О выполнении лабораторной работы №4**

**«Обработка одномерных массивов»**

Студент: Кафанов С.П.

Группа: Б21-515

Преподаватель: Храпов А.С.

Москва — 2021

**Задача. Вариант №5**

Необходимо спроектировать и реализовать на языке C программу, осуществляющую по запросам пользователя ввод, обработку и вывод последовательности данных, которая представляется в виде массива.

Структура «Автомобиль»:

• марка (строка длиной до 16 символов, которая может включать в себя только буквы, дефис и пробелы);

• ФИО владельца (строка произвольной длины);

• пробег (дробное число, соответсвующее величине пробега в тыс. км).

Сортировки

1. Гномья сортировка (Gnome sort).

2. Сортировка вставками с бинарным поиском (Insertion sort with binary search).

3. Поразрядная сортировка (Radix sort).

**Использованные типы данных.**

*Int –* для работы с простыми целочисленными данными

*Long double –* для поля пробег

*Size\_t –* для сортировок: арифметика указателя воид, хранение неопределённо большого числа

*Char –* для строк, чисел небольшого диапазона и работы с байтами памяти

*\* -* для работы с динамической памятью и передачи переменных в функцию, просто раоты с указателями

*Void \* -* для написания универсальных сортировок

**Код.**

Сортировки:

#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <string.h>  
#include "CarStruct.c"  
#include "Comporators.h"

void new\_swap(void \* a, void \* b, size\_t size) {  
 void \* buf = malloc(size);  
 memcpy(buf, a, size);  
 memcpy(a, b, size);  
 memcpy(b, buf, size);  
 free(buf);  
}  
  
void gnome\_sort(void \* first, size\_t number, size\_t size, int (\*comparator) (const void\*, const void\*)) {  
 void \* cars = first;  
  
 for (size\_t i = 1; i < number; ++i) {  
 for (size\_t j = i; j; --j) {  
 void \* p1 = (void\*) ((size\_t) cars + size\*(j - 1));  
 void \* p2 = (void\*) ((size\_t) cars + size\*j);  
 if (comparator(p1, p2) >= 0) {  
 new\_swap(p1, p2, size);  
 }  
 }  
 }  
}  
  
size\_t binarySearch(void \* cars, void \* el, size\_t number, size\_t size, int (\*comparator) (const void\*, const void\*)) {  
 size\_t left = 0, right = number;  
 while (left != right) {  
 size\_t m = (left+right)/2;  
 void \* p = (void\*) ((size\_t) cars + size\*m);  
 if (comparator(el, p) > 0) left = m + 1;  
 else if (comparator(el, p) < 0) right = m;  
 else return m;  
 }  
 return left;  
}  
  
// Функция для сортировки массива a [] размера 'n'  
void insertionSort(void \* cars, size\_t number, size\_t size, int (\*comparator) (const void\*, const void\*)) {  
 size\_t i, loc;  
 for (i = 1; i < number; ++i) {  
 void \* el = (void\*)((size\_t)cars+i\*size);  
 void \* buf = malloc(size);  
 memcpy(buf, el, size);  
 // найти место, где выбрано, может быть  
 loc = binarySearch(cars, el, i, size, comparator);  
  
 // Переместить все элементы после расположения, чтобы создать пространство  
 void\* donor = (void\*) ((size\_t)cars + (loc)\*size);//OK  
 void\* recipient = (void\*) ((size\_t)cars + (loc+1)\*size);//OK  
 size\_t len = i-loc;//OK  
 memmove(recipient, donor, len\*size);  
 memmove(donor, buf, size);  
 free(buf);  
 }  
}  
  
//должна возвращать указатель на индексованный байт с конца и размер элемента, по которому идёт  
// если индекс -1 то указатель на начало  
void\* el(void \* first, size\_t index, size\_t \* size\_ell, char \*\*\* el\_ptr, size\_t max\_len, char field) {  
 void \* ptr = NULL;  
 struct Car\* car = (struct Car\*)first;  
 switch (field) {  
 case 0:  
 \*size\_ell = 16;  
 if (index == -1)  
 ptr = (void\*)(car->mark);  
 else  
 ptr = (void\*)((size\_t)(car->mark)+(\*size\_ell- sizeof(char)\*(index+1)));  
 break;  
 case 2:  
 \*size\_ell = 10;//sizeof(long double)-6;  
 if (index == -1)  
 ptr = (void\*)&(car->mileage);  
 else  
 ptr = (void\*)((size\_t)&(car->mileage)+(sizeof(char)\*(index)));   
 break;  
 case 1: {  
 char \*\* p = &(car->name);  
 \*el\_ptr = p;  
  
 if (index == -1)  
 ptr = (void\*)(car->name);  
 else  
 ptr = (void\*)((size\_t)(car->name)+sizeof(char)\*(max\_len-(index+1)));   
 \*size\_ell = strlen(car->name)+1;  
 break;  
 }  
 default:  
 break;  
 }  
 return ptr;  
}  
  
void radix\_sort\_uni(void \* cars, size\_t number, size\_t size, char field) {  
 int sc = sizeof(char);  
 char \*\* buf = NULL;  
 //находим максимальную длину той памяти, которую будем сравнивать  
 size\_t max\_len = 0;  
 for (size\_t index = 0; index < number; ++index) {  
 size\_t len = 0;  
 void\* car = (void\*) ((size\_t)cars + index\*size);  
 void \* p = el(car, -1, &len, &buf, 0, field);  
 if (max\_len < len) max\_len = len;  
 }  
  
 // унификация длинн  
 for (size\_t i = 0; i < number; ++i) {  
 size\_t len = 0;  
 void \* car = (void\*) ((size\_t)cars + i\*size);  
 char \*\*p = NULL;  
 void \* byte = el(car, -1, &len, &p, max\_len, field);  
 if (len < max\_len) {  
  
 // нужен указатель на указатель на имя  
 // чтобы положить туда указатель на новую память  
 \*p = realloc(\*p, (1 + max\_len) \* sizeof(char));  
  
 int l = strlen(\*p);  
 for (int k = l; k <= max\_len; ++k) (\*p)[k] = '\0';  
 }  
 }  
  
 // основной цикл:  
 // - рассортировать по значению текущего байта  
 // - поменять массив  
 void \* copy\_arr = malloc(size\*number);  
 for (size\_t index = 0; index < max\_len; ++index) {  
 //корзинки для массивов указателей на указатели на элементы: ->|=>  
 void \*\*\* box = malloc(256\*sizeof(void\*\*));  
 for (size\_t i = 0; i < 256; ++i) {  
 box[i] = malloc(sizeof(void\*\*));  
 box[i][0] = NULL;  
 //\*((void\*\*\*)((size\_t)box+i\*sizeof(void\*\*))) = malloc(sizeof(void\*\*));  
 //\*\*((void\*\*\*)((size\_t)box+i\*sizeof(void\*\*))) = NULL;  
 }  
 memcpy(copy\_arr, cars, size\*number);  
  
 // разбрасывам указатели на элементы по корзинкам  
 for (size\_t i = 0; i < number; ++i) {  
 size\_t len = 0;  
 void \* car = (void\*) ((size\_t)copy\_arr + i\*size);  
 void \* byte = el(car, -1, &len, &buf, max\_len, field);  
  
 byte = el(car, index, &len, &buf, max\_len, field);  
 unsigned char n = \*((unsigned char\*)byte);  
  
 // смотрим сколько уже есть указателей  
 // и добавляем к ним ещё один  
 size\_t k = 0;  
 while (box[n][k] != NULL) k++;  
 box[n] = realloc(box[n], sizeof(void \*) \* (k + 2));  
 box[n][k + 1] = NULL;  
 box[n][k] = car;  
 }  
  
 int a = 0;  
 // переставляем индексы в массиве в соответствии с корзинками  
 for (size\_t i = 0; i < 256; ++i) {  
 int k = 0;  
 while (box[i][k] != NULL) {// вероятно ошибка при выделении памяти  
 memcpy((void\*)((size\_t)cars+a\*size), box[i][k], size); // ошибка индекса i  
 k++;  
 a++;  
 }  
 }  
 for (size\_t i = 0; i < 256; ++i) free(box[i]);  
 free(box);  
 }  
 free(copy\_arr);  
 //освобождаем нулёвую память  
 for (size\_t i = 0; i < number; ++i) {  
 size\_t len = 0;  
 void \* car = (void\*) ((size\_t)cars + i\*size);  
 void \* byte = el(car, -1, &len, &buf, max\_len, field);  
  
 if (len < max\_len) {  
 char \*\*p = NULL;  
 void \*buf1 = el(car, -1, &len, &p, 0, field);  
  
 // нужен указатель на указатель на имя  
 // чтобы положить туда указатель на новую память  
 \*p = realloc(\*p, (1 + strlen((char\*)buf1)) \* sizeof(char));  
 }  
 }  
}  
  
void sort(struct Car \*\* all\_cars, int number\_of\_cars, char index\_of\_sort) {  
 printf("With which characteristic do you wanna sort array?\n");  
 printf("1) Mark\n");  
 printf("2) Name\n");  
 printf("3) Mileage\n");  
 int chose = -1;  
 while (scanf("%d", &chose) <= 0) { getchar(); printf("Error!\n"); } getchar();  
 if (chose > 3 || chose < 1) {  
 printf("Wrong number. Try again.\n");  
 sleep(3);  
 return;  
 }  
 chose--;  
 int (\*comp[3])(const void\*, const void\*) = {comp\_mark, comp\_name, comp\_mileage};  
  
 switch(index\_of\_sort) {  
 case 0: gnome\_sort(\*all\_cars, number\_of\_cars, sizeof(struct Car), comp[chose]); break;  
 case 1: insertionSort(\*all\_cars, number\_of\_cars, sizeof(struct Car), comp[chose]); break;  
 case 2: radix\_sort\_uni(\*all\_cars, number\_of\_cars, sizeof(struct Car), (char) chose); break;  
 default: return;  
 }  
}

Меню:

#include <stdio.h>  
#include <stdlib.h>  
#include "MenuClass.c"  
#include <string.h>  
#include <unistd.h>

/\*

struct Folder{

char \* name;

struct Folder \*\* folders;

struct Folder \* previous;

char \*\* functions;

};

struct Menu{

struct Folder \* root;

struct Folder \*\* all\_folders;

int nof;

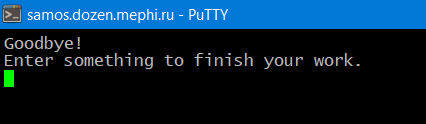
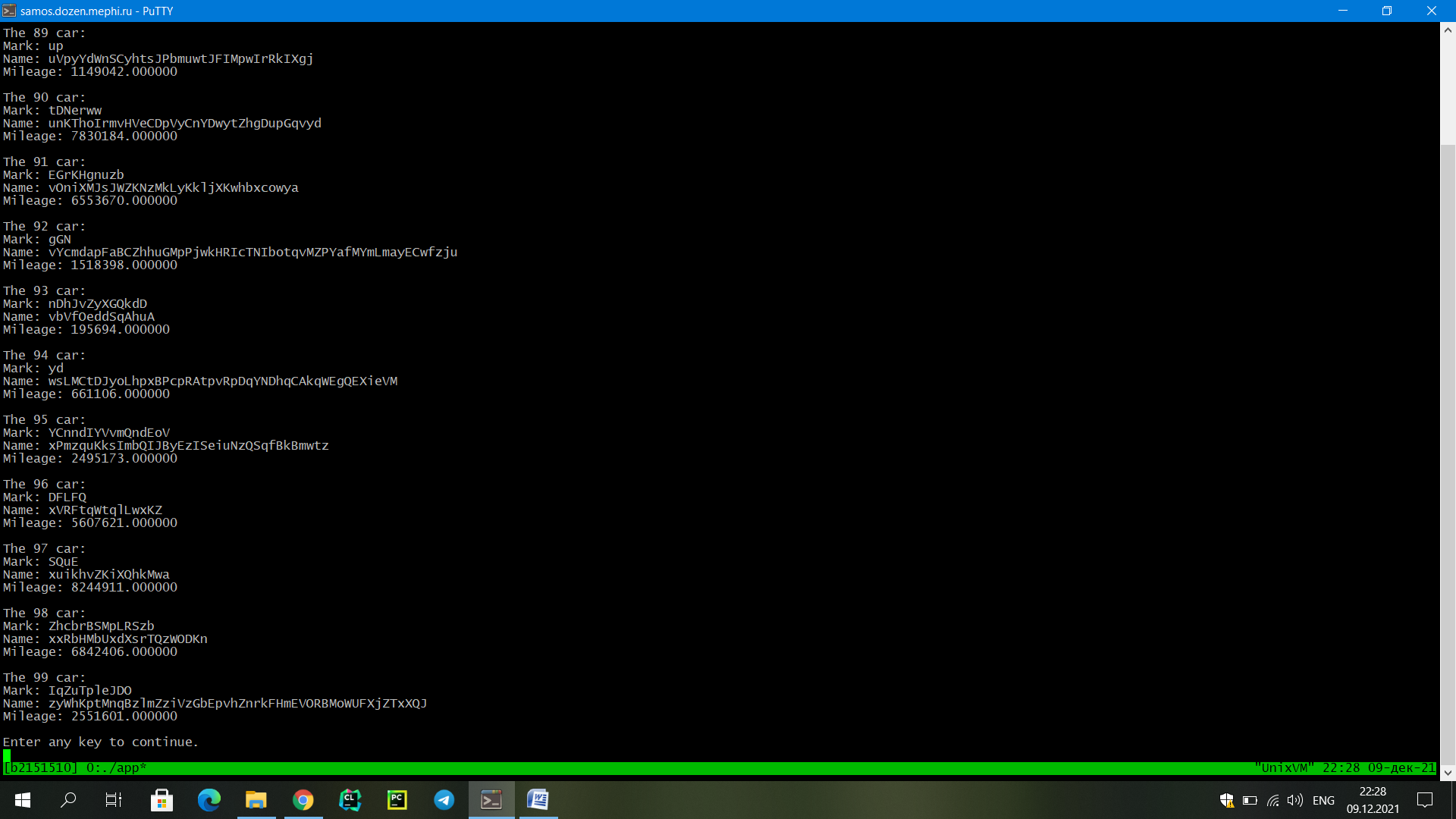
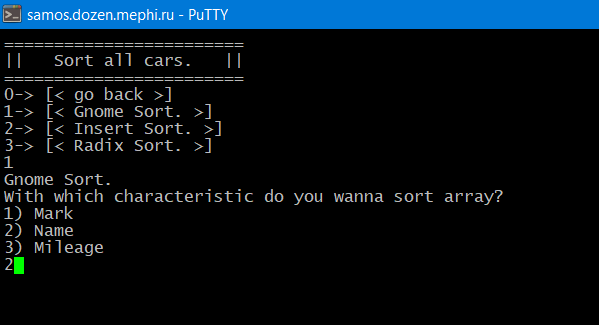
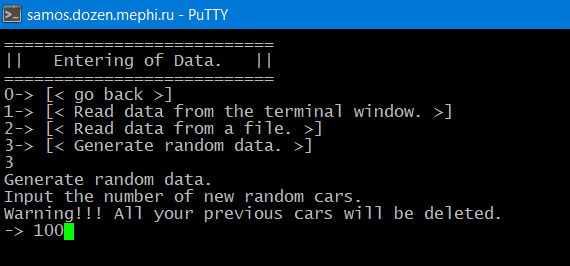
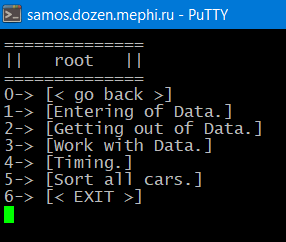
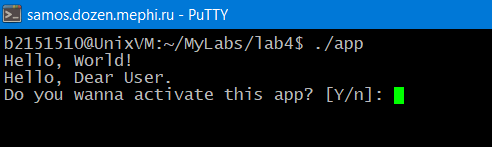
};\*/

struct Menu init\_menu() {  
 struct Menu menu;  
 menu.root = (struct Folder\*) malloc(sizeof(struct Folder));  
   
 menu.root->folders = (struct Folder\*\*) malloc(sizeof(struct Folder\*));  
 menu.root->folders[0] = NULL;  
   
 menu.root->previous = NULL;  
 //menu.name->name = (char\*) malloc(sizeof(char)\*5);  
   
 menu.root->name = (char\*) malloc(sizeof(char) \* strlen("root")+1);  
 memcpy(menu.root->name, "root", strlen("root")+1);//"root";  
   
 menu.root->functions = (char\*\*) malloc(sizeof(char\*)\*2);  
 //menu.root->functions[0] = (char\*) malloc(sizeof(char)\*5);  
 menu.root->functions[0] = "EXIT";  
 menu.root->functions[1] = NULL;  
   
 menu.all\_folders = (struct Folder\*\*) malloc(sizeof(struct Folder\*));  
 menu.all\_folders[0] = NULL;  
   
 menu.nof = 0;  
  
 return menu;  
}  
  
int number\_of\_folders(struct Folder folder) {  
 int i = 0;  
 while (folder.folders[i]) i++;  
 return i;  
}  
  
int number\_of\_functions(struct Folder folder) {  
 int i = 0;  
 while (folder.functions[i]) i++;  
 return i;  
}  
  
void add\_folder(char \* name, char \* host\_name, struct Menu \* menu) {  
 struct Folder \* f = menu->root;  
 if (menu->nof == 0){   
 f->folders[0] = (struct Folder\*) malloc(sizeof(struct Folder));  
  
 f->folders[0]->previous = f;  
  
 f = f->folders[0];  
  
 f->name = (char\*) malloc(sizeof(char) \* strlen(name)+1);  
 memcpy(f->name, name, strlen(name)+1);  
  
 f->folders = (struct Folder\*\*) malloc(sizeof(struct Folder\*));  
 f->folders[0] = NULL;  
   
 f->functions = (char\*\*) malloc(sizeof(char\*));  
 f->functions[0] = NULL;  
  
 int nof = menu->nof;  
 menu->all\_folders = (struct Folder \*\*) realloc(menu->all\_folders, sizeof(struct Folder\*) \* (nof + 2));  
 menu->all\_folders[nof] = f;  
 menu->all\_folders[nof+1] = NULL;  
 menu->nof++;  
 }  
 else {  
 int nof = menu->nof;  
 for (int i = 0; i < nof; ++i) {  
 if (!strcmp(menu->all\_folders[i]->name, host\_name)) {  
 f = menu->all\_folders[i];  
 int nof1 = number\_of\_folders(\*f);  
   
 f->folders = (struct Folder\*\*) realloc(f->folders, (nof1+2) \* sizeof(struct Folder\*));  
  
 f->folders[nof1+1] = NULL;  
 f->folders[nof1] = (struct Folder\*) malloc(sizeof(struct Folder));  
 f->folders[nof1]->previous = f;  
   
 f = f->folders[nof1];  
  
 f->name = (char\*) malloc(sizeof(char) \* strlen(name)+1);  
 memcpy(f->name, name, strlen(name)+1);  
  
 f->folders = (struct Folder\*\*) malloc(sizeof(struct Folder\*));  
 f->folders[0] = NULL;  
  
 f->functions = (char\*\*) malloc(sizeof(char\*));  
 f->functions[0] = NULL;  
   
 menu->all\_folders = (struct Folder \*\*) realloc(menu->all\_folders, sizeof(struct Folder\*) \* (nof + 2));  
 menu->all\_folders[nof] = f;  
 menu->all\_folders[nof+1] = NULL;  
 menu->nof++;  
 break;  
 }  
 }  
 if (!strcmp(host\_name, menu->root->name)) {  
 f = menu->root;  
  
 f->folders = (struct Folder\*\*) realloc(f->folders, (nof+2) \* sizeof(struct Folder\*));  
  
 f->folders[nof+1] = NULL;  
 f->folders[nof] = (struct Folder\*) malloc(sizeof(struct Folder));  
 f->folders[nof]->previous = f;  
  
 f = f->folders[nof];  
  
 f->name = (char\*) malloc(sizeof(char) \* strlen(name)+1);  
 memcpy(f->name, name, strlen(name)+1);  
  
 f->folders = (struct Folder\*\*) malloc(sizeof(struct Folder\*));  
 f->folders[0] = NULL;  
  
 f->functions = (char\*\*) malloc(sizeof(char\*));  
 f->functions[0] = NULL;  
  
 menu->all\_folders = (struct Folder \*\*) realloc(menu->all\_folders, sizeof(struct Folder\*) \* (nof + 2));  
 menu->all\_folders[nof] = f;  
 menu->all\_folders[nof+1] = NULL;  
 menu->nof++;  
 }  
 }  
}  
  
void add\_function(char \* name, char \* host\_name, struct Menu menu) {  
 int nof = menu.nof;  
 for (int i = 0; i < nof; ++i) {  
  
 if (!strcmp(menu.all\_folders[i]->name, host\_name)) {  
 struct Folder \* f = menu.all\_folders[i];  
 int nof1 = number\_of\_functions(\*f);  
 f->functions = (char\*\*) realloc(f->functions, sizeof(char\*) \* (nof1+2));  
 f->functions[nof1+1] = NULL;  
  
 f->functions[nof1] = (char\*) malloc(sizeof(char) \* (strlen(name) + 1));  
 memcpy(f->functions[nof1], name, strlen(name));  
 f->functions[nof1][strlen(name)] = '\0';  
  
 break;  
 }  
 }  
}  
  
void delete\_all\_folders(struct Menu \* menu) {  
 for (int i = 0; i < menu->nof; ++i) {  
 struct Folder \* f = menu->all\_folders[i];  
 free(f->folders);  
 free(f->name);  
   
 for (int i = 0, l = number\_of\_functions(\*f); i < l; ++i) {  
 free(f->functions[i]);  
 }  
 free(f->functions);  
 free(f);  
 }  
}  
  
void clear\_root(struct Menu \* menu) {  
 free(menu->root->name);  
 free(menu->root->folders);  
 //я делал указатель на статическую память  
 //free(\*(menu->root->functions));  
 free(menu->root->functions);  
 free(menu->root);  
 free(menu->all\_folders);  
}  
  
struct Menu create\_menu() {  
 struct Menu menu = init\_menu();  
 char enter\_data[] = "Entering of Data.";  
 char get\_out\_data[] = "Getting out of Data.";  
 char work\_with\_data[] = "Work with Data.";  
 char data\_time[] = "Timing.";  
 char root[] = "root";  
 char sort[] = "Sort all cars.";  
  
 char read\_from\_terminal[] = "Read data from the terminal window.";  
 char read\_from\_file[] = "Read data from a file.";  
 char random\_generation[] = "Generate random data.";  
 char output\_to\_terminal[] = "Print cars to terminal.";  
 char save\_data\_to\_file[] = "Save data to file.";  
 char insert[] = "Insert a new car.";  
 char insert\_in\_sorted\_array[] = "Insert a new car in the sorted array.";  
 char erase[] = "Delete several elements.";  
 char timing[] = "Timing.";  
 char gnome\_sort[] = "Gnome Sort.";  
 char insert\_binary\_sort[] = "Insert Sort.";  
 char radix\_sort[] = "Radix Sort.";  
   
 add\_folder(enter\_data, root, &menu);  
 add\_folder(get\_out\_data, root, &menu);  
 add\_folder(work\_with\_data, root, &menu);  
 add\_folder(data\_time, root, &menu);  
 add\_folder(sort, root, &menu);  
   
 add\_function(read\_from\_terminal, enter\_data, menu);  
 add\_function(read\_from\_file, enter\_data, menu);  
 add\_function(random\_generation, enter\_data, menu);  
   
 add\_function(output\_to\_terminal, get\_out\_data, menu);  
 add\_function(save\_data\_to\_file, get\_out\_data, menu);  
  
 add\_function(insert, work\_with\_data, menu);  
 add\_function(insert\_in\_sorted\_array, work\_with\_data, menu);  
 add\_function(erase, work\_with\_data, menu);  
  
 add\_function(gnome\_sort, sort, menu);  
 add\_function(insert\_binary\_sort, sort, menu);  
 add\_function(radix\_sort, sort, menu);  
  
 add\_function(timing, data\_time, menu);  
  
 return menu;  
}  
  
void print\_menu(struct Folder \* f) {  
 for (int i = 0; i < strlen(f->name)+10; ++i) printf("="); printf("\n");  
 printf("|| %s ||\n", f->name);  
 for (int i = 0; i < strlen(f->name)+10; ++i) printf("="); printf("\n");  
  
 printf("0-> [< go back >]\n");  
  
 int i = 0;  
 while (f->folders[i++])  
 printf("%d-> [%s]\n", i, f->folders[i-1]->name);  
  
 int j = 0;  
 while (f->functions[j++])  
 printf("%d-> [< %s >]\n", i+j-1 , f->functions[j-1]);  
}

**Тесты.**

Отсортировать массив из 100 сгенерированных машин по именам владельцев, используя Gnome Sort.

**Скриншоты.**



**Таймирование.**

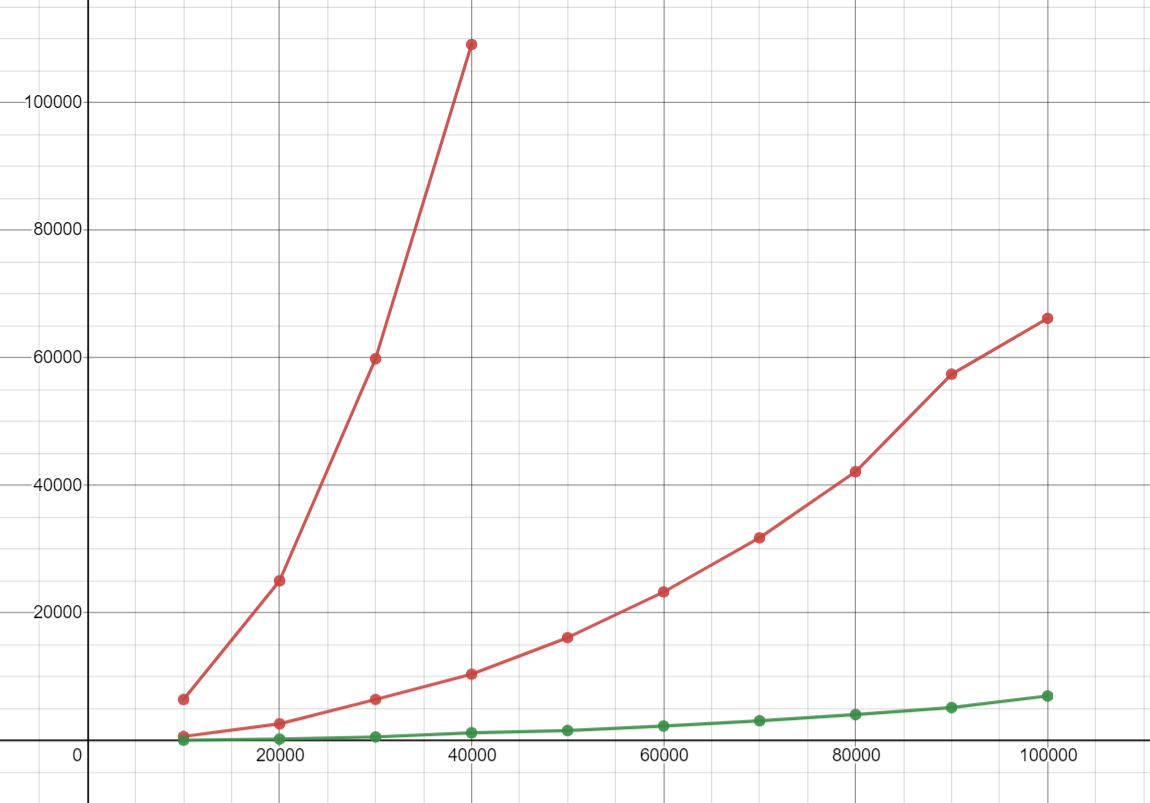
Таймирование производилось в «тиках» для пущей наглядности.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Gnome Sort | Insert Sort with binary search | Radix sort (LSD) |
| Время на поле «пробег»  10000 cars  20000 cars  30000 cars  40000 cars  50000 cars  60000 cars  70000 cars  80000 cars  90000 cars  100000 cars | 6090  23306  52737  97561  150734  >150000 | 67  256  578  1019  1596  2337  3504  4272  5697  7212 | 652  2640  6462  10415  16149  23307  31783  42148  57430  66186 |

Асимптотики сортировок в соответствующем таблице порядке:

* O(n2)
* O(n\*log(n))
* O(k\*n) – где k – это разрядность сортируемой памяти. В моём конкретном случае – количество байт, отводимых под запись каждого сортируемого элемента.

**Графики.**



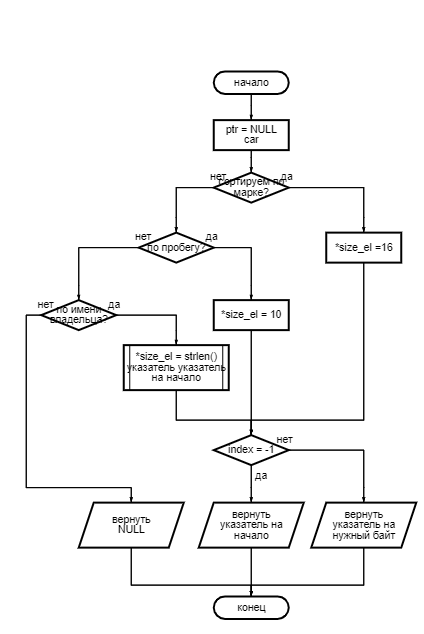
Gnome Sort

Radix Sort

Insert Sort

**Блок схема функций обработки.**

A B C D

**void\* el(void \* first, size\_t index, size\_t \* size\_ell, char \*\*\* el\_ptr, size\_t max\_len, char field)**

1

2

3

4

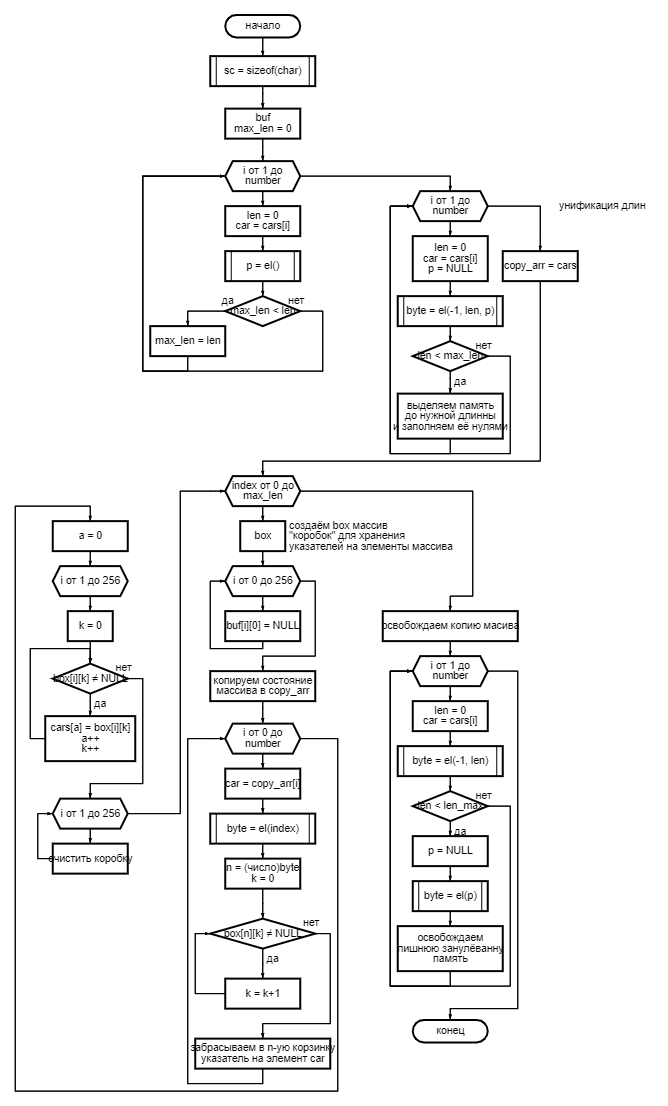
5

6

7

8

9



A B C D E

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

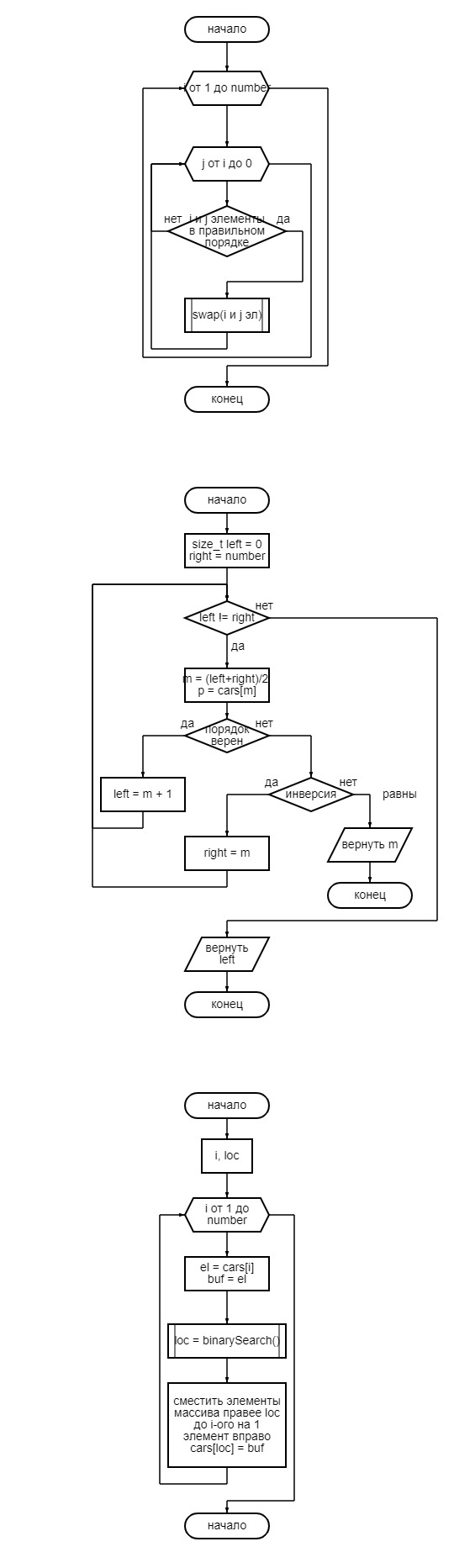
19

20

21

22

**void radix\_sort\_uni(void \* cars, size\_t number, size\_t size, char field)**



Gnome Sort

Binary Search

Insert Sort with binary search

A B C D

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

**Сравнительный анализ.**

По результатам проведённого таймирования лучшие результаты показала сортировка с бинарными вставками.

При этом лучшая из представленных по асимптотике поразрядная сортировка оказалась лишь на 2-ом месте показав результаты примерно в 10 раз худшие.

Вероятнее всего, это произошло по следующим причинам:

* Количество данных относительно не велико.
* По своей задумке и идее Radix – специализированная сортировка, поскольку работает напрямую с байтами. Но из-за условий лабораторной работы пришлось сделать её универсальной для любого набора данных, что привело к значительному падению как эффективности в целом, так и скорости в частности.

**Выводы:**

В ходе данной лабораторной работы:

* Были закреплены навыки работы с динамической памятью и указателями.
* Я научился работать с указателями типа void\* и типом данных size\_t.
* Научился азам работы с Makefile-ом.
* Написал универсальную структуру меню: «аля» каталог файлов с функциями в них, которую можно будет использовать в дальнейшем.
* Написаны 3 универсальные сортировки разной асимптотики.
* Получен опыт работы с массивами структур и структурами в принципе.