Национальный исследовательский ядерный университет «МИФИ»

Институт интеллектуальных кибернетических систем

Кафедра №12 «Компьютерные системы и технологии»

ОТЧЕТ

О выполнении лабораторной работы №5 «Работа с массивами структур. Исследование методов сортировки массивов»

Студент: Рыженко Р.В.

Группа: Б23-506

Преподаватель: Курочкина М-А.А.

Москва 2023

1. Формулировка индивидуального задания

Вариант №364.

Индивидуальное задание Структура данных Объект недвижимости:

- адрес (строка произвольной длины);
- кадастровый номер (строка длиной 11 символов формата XX:YY:00:XX), где X — цифра, YY — буква;
- площадь (дробное число).

Алгоритмы сортировки

- Сортировка расчёской (Comb sort).
- Пирамидальная сортировка (Heap sort).

2. Описание использованных типов данных

При выполнении данной лабораторной работы использовался встроенные типы данных double и int, предназначенные для работы с вещественными и целыми числами, а также char для работы с символами и строками и указатели, предназначенные для работы с адресами в памяти. Также созданы собственные типы данных Item, хранящий в себе адрес и номер – указатели на строки, и площадь – дробное число, и List, хранящий указатель на Item.

3. Описание использованного алгоритма

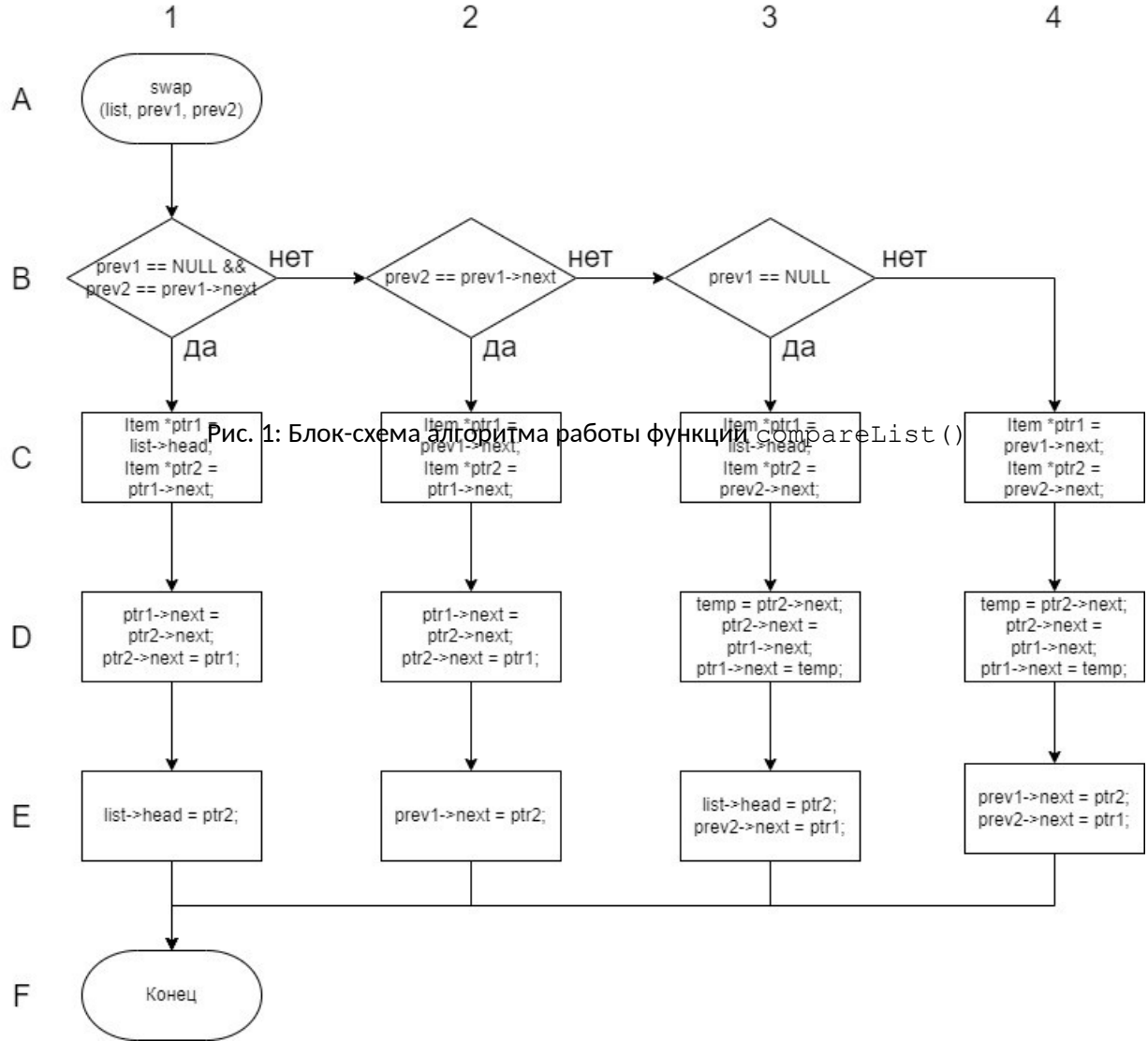


Рис. 1: Блок-схема алгоритма работы функции `compareList()`

Рис. 2: Блок-схема алгоритма работы функции `swap()`

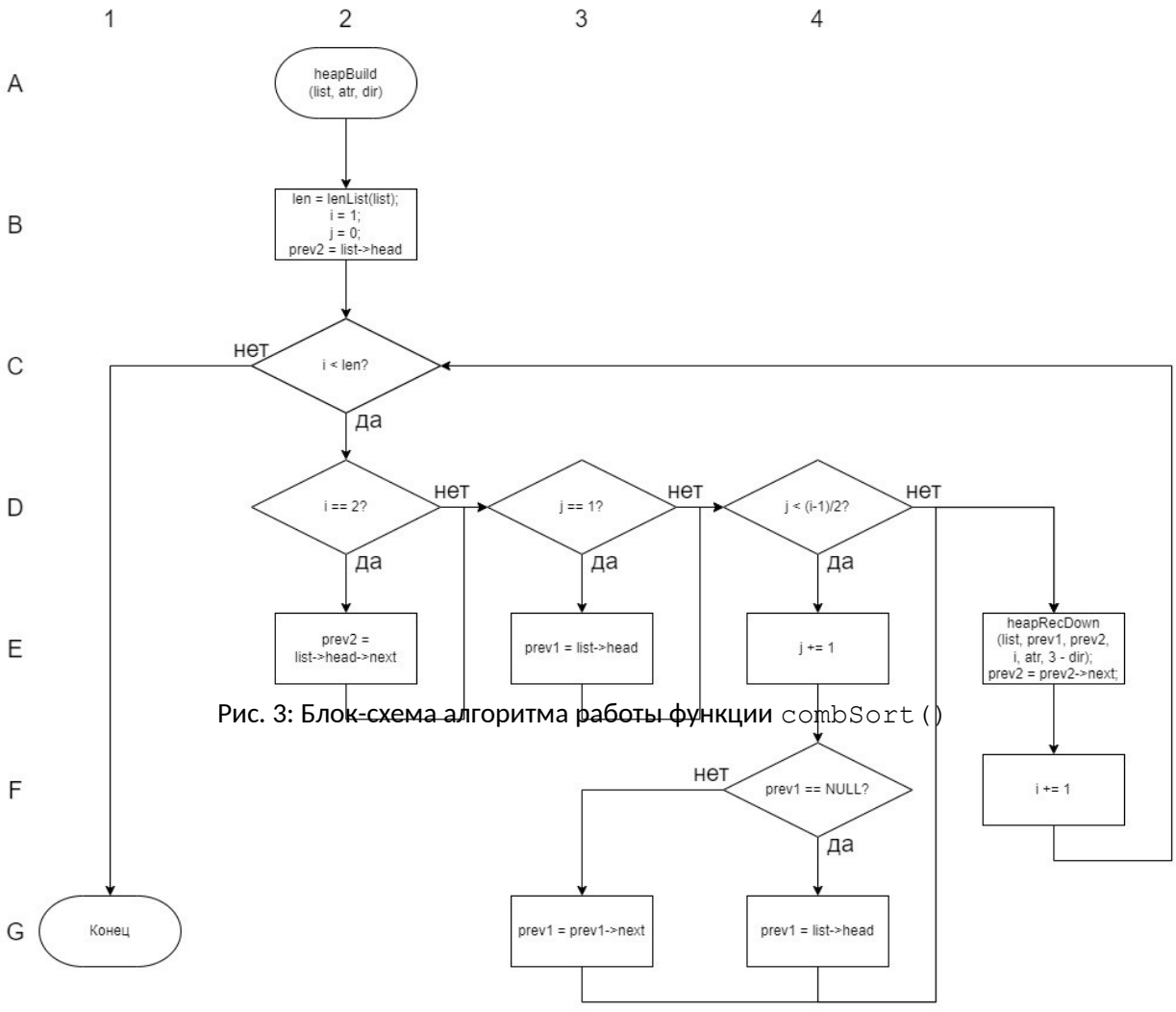


Рис. 3: Блок-схема алгоритма работы функции `combSort()`

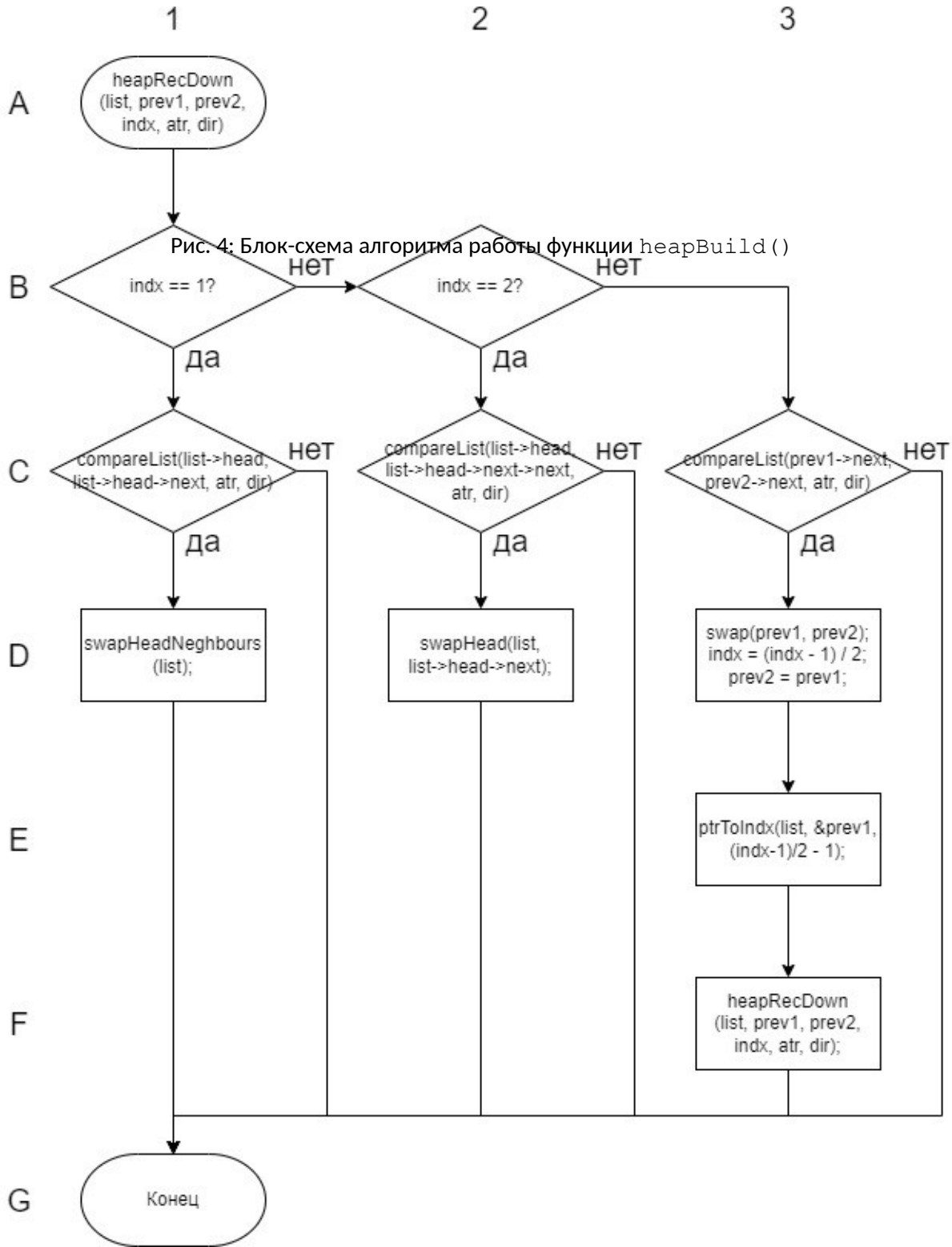


Рис. 4: Блок-схема алгоритма работы функции `heapBuild()`

Рис. 5: Блок-схема алгоритма работы функции `heapRecDown()`

Рис. 6: Блок-схема алгоритма работы функции `heapSort`

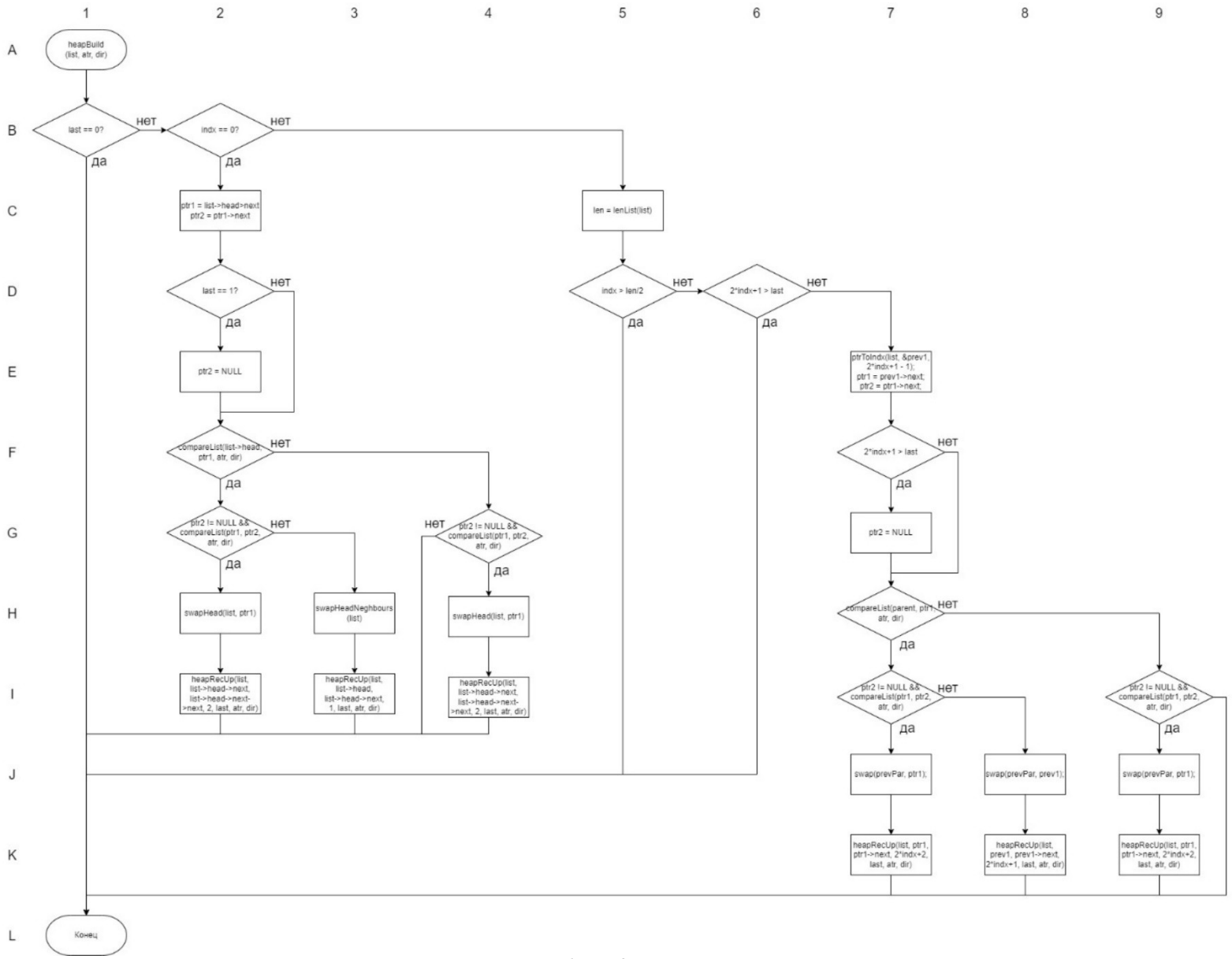


Рис. 7: Блок-схема алгоритма работы функции `heapRecUp()`

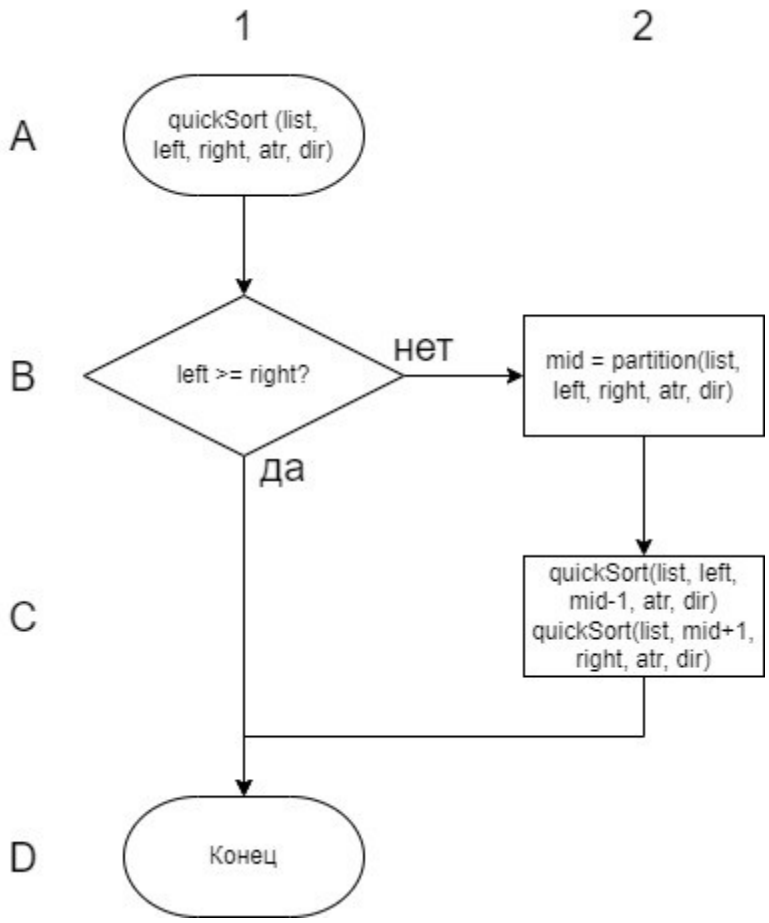


Рис. 8: Блок-схема алгоритма работы функции `quickSort()`

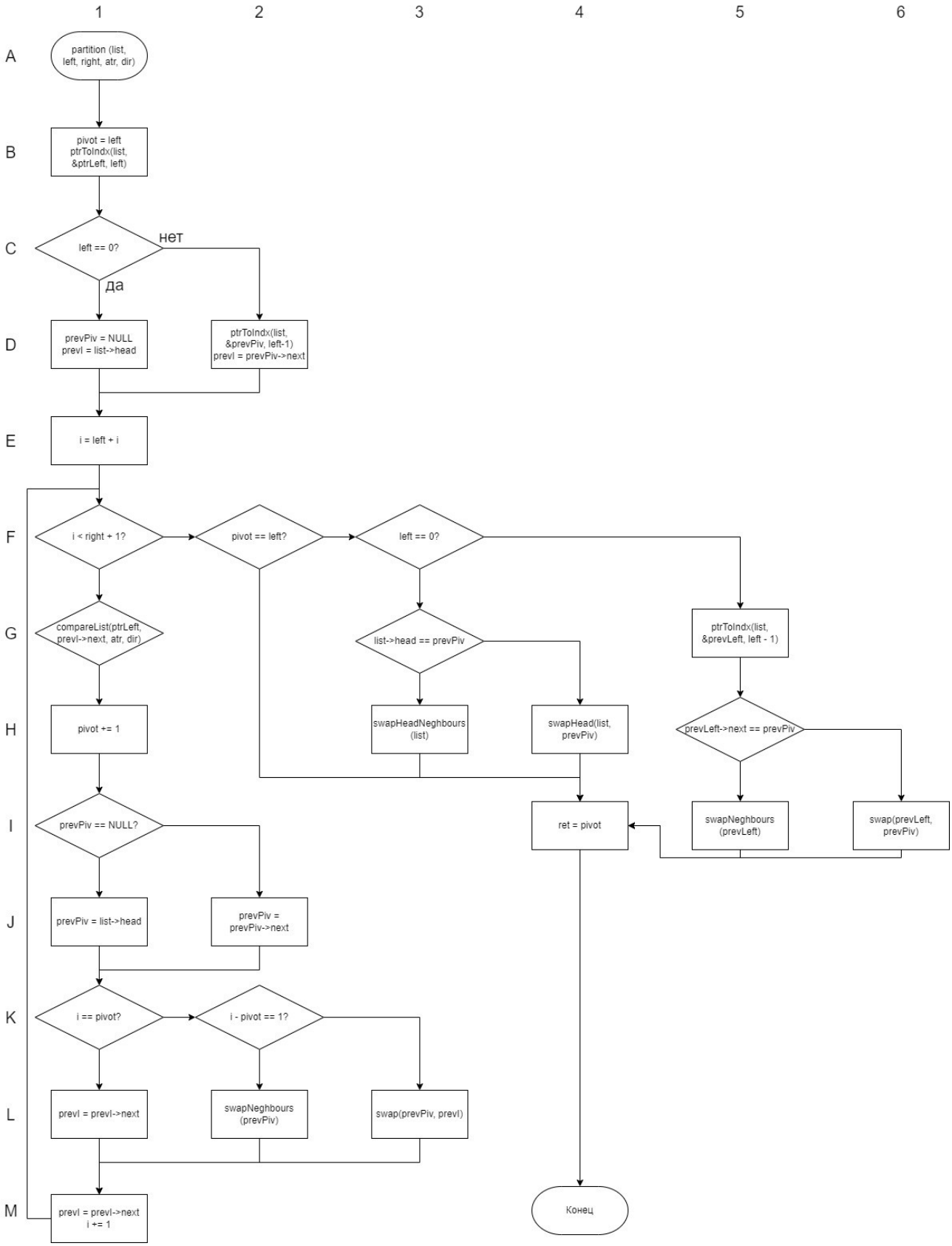


Рис. 8: Блок-схема алгоритма работы функции `partition()`

4. Исходные коды разработанных программ

Листинг 1: Исходный код программы 1 – файл main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "other.h"

int main()
{
   List *list = (List*) malloc(sizeof(List));
   if (list == NULL) {
      printf("Не найдено свободной памяти\n");
      return 0;
   }
   list->head = NULL;

   int fl = 1;
   int panel;
   int pS1, pS2, pS3; // panelSort, три выбора там
   int returned;
   while (fl == 1) {
      printf("Выберите одну из опций:\n\
         (1) Ввод списка\n\
         (2) Вывод списка\n\
         (3) Сортировка списка\n\
         (4) Завершение программы\n");

      panel = 0;
      while (panel < 1  ||  panel > 4) {
         printf("Введите число от 1 до 4!\n");
         returned = safeScanfInt(&panel);
         if (returned == 0) {
            fl = 0;
            endOfProgram(list);
            return 0;
         }
      }

      switch(panel) {
      case 1: // ввод списка
         returned = initializeList(list);
         if (returned == 0) {
            fl = 0;
            endOfProgram(list);
            return 0;
         }

         printf("Список создан\n");
         returned = outputList(list);
         if (returned == 0) {
            fl = 0;
            endOfProgram(list);
            return 0;
         }
         break;

      case 2: // вывод списка
         returned = outputList(list);
         if (returned == 0) {
            fl = 0;
            endOfProgram(list);
            return 0;
         }
         break;

      case 3: // сортировка списка
         printf("Выберите алгоритм сортировки:\n\
            (1) Сортировка расчёской (Comb sort)\n\
            (2) Пирамидальная сортировка (Heap sort)\n\
            (3) Быстрая сортировка (qsort)\n");
         pS1 = -1;
         while (pS1 < 1 || pS1 > 3) {
            printf("Введите целое число от 1 до 3!\n");
```

```
            returned = safeScanfInt(&pS1);
            if (returned == 0) {
                fl = 0;
                endOfProgram(list);
                return 0;
            }
        }

        printf("Выберите поле структуры, по которому будем сортировать:\n\
            (1) Адрес\n\
            (2) Кадастровый номер\n\
            (3) Площадь\n");
        pS2 = -1;
        while (pS2 < 1 || pS2 > 3) {
            printf("Введите целое число от 1 до 3!\n");
            returned = safeScanfInt(&pS2);
            if (returned == 0) {
                fl = 0;
                endOfProgram(list);
                return 0;
            }
        }

        printf("Выберите направление сортировки (возрастание/убывание:\n\
            (1) Возрастание\n\
            (2) Убывание\n");
        pS3 = -1;
        while (pS3 < 1 || pS3 > 2) {
            printf("Введите целое число от 1 до 2!\n");
            returned = safeScanfInt(&pS3);
            if (returned == 0) {
                fl = 0;
                endOfProgram(list);
                return 0;
            }
        }

        switch(pS1) {
        case 1:
            combSort(list, pS2, pS3);
            break;
        case 2:
            heapSort(list, pS2, pS3);
            break;
        case 3:
            int lenlist = lenList(list);
            quickSort(list, 0, lenlist-1, pS2, pS3);
            break;
        }
        printf("Список отсортирован\n");
        returned = outputList(list);
        if (returned == 0) {
            fl = 0;
            endOfProgram(list);
            return 0;
        }
        break;

    case 4: // завершение программы
        fl = 0;
        endOfProgram(list);
        return 0;
    }
  }

  // перебирает все элементы и каждый free(), включая поля элемента, а затем и сам list
  endOfProgram(list); // с принтом "Завершение программы\n"
  return 0;
}
```

Листинг 2: Исходный код программы 1 – файл other.h

```
#ifndef OTHER_H
#define OTHER_H

typedef struct Item {
   char *adr, *num;
   double sqr;
   struct Item *next;
} Item;

typedef struct List {
   Item *head;
} List;

int safeScanfInt (int *target);
int safeScanfDouble (double *target);
int safeScanfNum (char *target);
int isValidNum (char *string);
int readlFile (char **target, FILE *filePointer);
int safeFileScanfInt (int *target, FILE *filePointer);
int safeFileScanfDouble (double *target, FILE *filePointer);
int readlBinary (char **target, FILE *filePointer);
int safeBinaryScanfInt (int *target, FILE *filePointer);
int safeBinaryScanfDouble (double *target, FILE *filePointer);

int initializeList (List *list);
int outputList (List *list);
int lenList (List *list);
void ptrToIndx (List *list, Item **ptr, int indx);

void combSort (List *list, int atr, int dir);
void heapRecDown (List *list, Item *prev1, Item *prev2, int indx, int atr, int dir);
void heapRecUp (List *list, Item *prevPar, Item *parent, int indx, int last, int atr, int dir);
void heapBuild (List *list, int atr, int dir);
void heapSort (List *list, int atr, int dir);
void quickSort (List *list, int left, int right, int atr, int dir);
int partition (List *list, int left, int right, int atr, int dir);
int compareList (Item *item1, Item *item2, int atr, int dir);
void swap (Item *prev1, Item *prev2);
void swapNeghbours (Item *prev1);
void swapHead (List *list, Item *prev2);
void swapHeadNeghbours (List *list);

void endOfProgram (List *list);
void freeList(List *list);

#endif
```

Листинг 3: Исходный код программы 1 – файл other.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// #include <readline/readline.h>
#include "other.h"

void freeList(List *list)
{
   Item *ptr = list->head;
   while (ptr != NULL) {
         Item *next = ptr->next;
         free(ptr->adr);
         free(ptr->num);
         free(ptr);
         ptr = next;
   }
}

void endOfProgram (List *list)
{
   freeList(list);
   free(list);
   printf("Завершение программы\n");
}

char *readl(char *s)
{
```

```
    printf("%s", s);

    char *ptr = (char*)malloc(1);
    if (ptr == NULL) {
        printf("Не найдено свободной памяти\n");
        return NULL;
    }
    *ptr = '\0';
    char buf[81];
    int n=0, len=0;
    do {
        n = scanf("%80[^\n]", buf);
        if (n < 0) {
            free(ptr);
            ptr = NULL;
            continue;
        }
        if (n == 0) {
            scanf("%*c");
        }
        else {
            len += strlen(buf);
            char *temp = (char*)realloc(ptr, (len+1)*sizeof(char));
            if (temp == NULL) {
                free(ptr);
                return "\0";
            }
            ptr = temp;
            strcat(ptr, buf);
        }
    } while (n > 0);

    return ptr;
}

int safeScanfInt (int *target)
{
    int guard;
    int flag = 1;
    while (flag == 1) {
        guard = scanf("%d",target);
        scanf("%*[^\n]");
        scanf("%*c");
        if (guard == EOF) {
            return 0;
        }
        if (guard < 1) {
            printf("Введите целое число!\n");
            continue;
        }
        flag = 0;
    }
    return 1;
}

int safeScanfDouble (double *target)
{
    int guard;
    int flag = 1;
    while (flag == 1) {
        guard = scanf("%lf",target);
        scanf("%*[^\n]");
        scanf("%*c");
        if (guard == EOF) {
            return 0;
        }
        if (guard < 1) {
            printf("Введите число!\n");
            continue;
        }
        flag = 0;
    }
    return 1;
}

int safeScanfNum (char *target) // XX:YY:00:XX
{
    int valid = 0;
    char *s;
```

```
    while (valid == 0) {
        printf("Введите строку по формату XX:YY:00:XX, где X-цифра 0-9, Y - буква\n");
        s = readl("");
        valid = isValidNum(s);
        if (valid == -1) {
            free(s);
            return 0;
        }
    }
    strcpy(target, s);
    free(s);
    return 1;
}

int isValidNum (char *str)
{
    if (str == NULL || strlen(str) == 0) {
        return -1;
    }
    if (strlen(str) != 11) {
        return 0;
    }
    int i;
    for (i = 0; i < 11; i++) {
        if (i==2 || i==5 || i==8) {
            if (str[i] != ':') {
                return 0;
            }
        }
        else if (i==3 || i==4) {
            if (str[i]<'A' || str[i]>'z' || (str[i]>'Z' && str[i]<'a')) {
                return 0;
            }
        }
        else if (i==6 || i==7) {
            if (str[i] != '0') {
                return 0;
            }
        }
        else {
            if (str[i]<'0' || str[i]>'9') {
                return 0;
            }
        }
    }
    return 1;
}

int readlFile (char **target, FILE *filePointer)
{
    char *temp;
    temp = (char*) malloc(1*sizeof(char));
    if (temp == NULL) {
        printf("Не найдено свободной памяти\n");
        return 0;
    }
    *target = temp;
    *(target)[0] = '\0';
    char buf[10];
    int len, offset;
    char *spacePos;
    int fl = 1;
    while (fgets(buf, sizeof(buf), filePointer) != NULL) {
        if (buf[strlen(buf) - 1] == '\n') {
            buf[strlen(buf) - 1] = '\0';
            fl = 0;
            if (strlen(buf) == 0) {
                fl = 1; // если это самый первый считанный символ, то продолжаем
                continue;
            }
        }

        spacePos = strchr(buf, ' ');
        if (spacePos != NULL) {
            *spacePos = '\0';
            // девять, т.к размер buf = 10, но последний - \0
            // и ещё -1, т.к 9-len сдвинет снова на пробел, а надо сразу после него
            offset = 9 - strlen(buf) - 1;
            fseek(filePointer, 0-offset, SEEK_CUR);
```

```
            fl = 0;
            if (strlen(buf) == 0) {
                fl = 1; // если это самый первый считанный символ, то продолжаем
                continue;
            }
        }

        len = strlen(*target) + strlen(buf) + 1;

        temp = (char*) realloc(*target, len*sizeof(char)); // всегда realloc, т.к уже объявили на 1 эл.
        if (temp == NULL) {
            printf("Не найдено свободной памяти\n");
            // не делаю free(target);, т.к это будет во внешней функции, откуда target пришёл
            // файл не закрываю по той же причине
            return 0;
        }

        *target = temp;
        strcat(*target, buf);

        if (fl == 0) {
            break;
        }
    }
    return 1;
}

int safeFileScanfInt (int *target, FILE *filePointer)
{
    char a;
    fscanf(filePointer, "%c", &a);
    while (a == ' ' || a == '\n') {
        fscanf(filePointer, "%c", &a);
    }
    fseek(filePointer, -1, SEEK_CUR); // на 1 назад, т.к последний считанный - валидный

    int guard = fscanf(filePointer, "%d", target);
    fseek(filePointer, 1, SEEK_CUR); // pass всякие ' ' и \n
    // fscanf(filePointer, "%*c");
    if (guard == EOF) {
        return -1;
    }
    if (guard < 1) {
        return 0;
    }
    return 1;
}

int safeFileScanfDouble (double *target, FILE *filePointer)
{
    char a;
    fscanf(filePointer, "%c", &a);
    while (a == ' ' || a == '\n') {
        fscanf(filePointer, "%c", &a);
    }
    fseek(filePointer, -1, SEEK_CUR); // на 1 назад, т.к последний считанный - валидный

    int guard = fscanf(filePointer, "%lf", target);
    fseek(filePointer, 1, SEEK_CUR); // pass всякие ' ' и \n
    // fscanf(filePointer, "%*c");
    if (guard == EOF) {
        return -1;
    }
    if (guard < 1) {
        return 0;
    }
    return 1;
}

int readlBinary (char **target, FILE *filePointer)
{
    char *temp;
    temp = (char*) malloc(1*sizeof(char));
    if (temp == NULL) {
        printf("Не найдено свободной памяти\n");
        return 0;
    }
    *target = temp;
    *(target)[0] = '\0';
```

```
    char buf[10];
    buf[9] = '\0';

    int len, offset;
    char *spacePos, *enterPos;
    int fl = 1;
    while (fread(buf, sizeof(char), 9, filePointer) > 0) {
        spacePos = strchr(buf, ' ');
        enterPos = strchr(buf, '\n');
        if (spacePos != NULL && enterPos != NULL) {
            if (spacePos < enterPos) {
                enterPos = NULL; // оставляем меньший указатель
            }
            else {
                spacePos = NULL;
            }
        }
        if (spacePos != NULL) {
            *spacePos = '\0';
            // девять, т.к размер buf = 10, но последний - \0
            // и ещё -1, т.к 9-len сдвинет снова на пробел, а надо сразу после него
            offset = 9 - strlen(buf) - 1;
            fseek(filePointer, 0-offset, SEEK_CUR);
            fl = 0;
            if (strlen(buf) == 0) {
                fl = 1; // если это самый первый считанный символ, то продолжаем
                continue;
            }
        }
        if (enterPos != NULL) {
            *enterPos = '\0';
            offset = 9 - strlen(buf) - 1;
            fseek(filePointer, 0-offset, SEEK_CUR);
            fl = 0;
            if (strlen(buf) == 0) {
                fl = 1; // если это самый первый считанный символ, то продолжаем
                continue;
            }
        }


        len = strlen(*target) + strlen(buf) + 1;
        temp = (char*) realloc(*target, len*sizeof(char)); // всегда realloc, т.к уже объявили на 1 эл.
        if (temp == NULL) {
            printf("Не найдено свободной памяти\n");
            // не делаю free(target);, т.к это будет во внешней функции, откуда target пришёл
            // файл не закрываю по той же причине
            return 0;
        }

        *target = temp;
        strcat(*target, buf);

        if (fl == 0) {
            break;
        }
    }
    return 1;
}

int safeBinaryScanfInt (int *target, FILE *filePointer)
{
    char a;
    fread(&a, sizeof(char), 1, filePointer);
    while (a == ' ' || a == '\n') {
        fread(&a, sizeof(char), 1, filePointer);
    }
    fseek(filePointer, -1, SEEK_CUR); // на 1 назад, т.к последний считанный - валидный

    int guard = fread(target, sizeof(int), 1, filePointer);
    fseek(filePointer, 1, SEEK_CUR); // pass всякие ' ' и \n
    if (guard == EOF) {
        return -1;
    }
    if (guard < 1) {
        return 0;
    }
    return 1;
}
```

```
int safeBinaryScanfDouble (double *target, FILE *filePointer)
{
    char a;
    fread(&a, sizeof(char), 1, filePointer);
    while (a == ' ' || a == '\n') {
        fread(&a, sizeof(char), 1, filePointer);
    }
    fseek(filePointer, -1, SEEK_CUR); // на 1 назад, т.к последний считанный - валидный

    int guard = fread(target, sizeof(double), 1, filePointer);
    fseek(filePointer, 1, SEEK_CUR); // pass всякие ' ' и \n
    if (guard == EOF) {
        return -1;
    }
    if (guard < 1) {
        return 0;
    }
    return 1;
}



int initializeList (List *list)
{
    freeList(list);
    printf("Выберите, откуда осуществлять ввод:\n\
        (1) из стандартного потока ввода потока («с клавиатуры»)\n\
        (2) из текстового файла\n\
        (3) из бинарного файла\n");
    int panFile = 0;
    int returned;
    while (panFile < 1  ||  panFile > 3) {
        printf("Введите число от 1 до 3!\n");
        returned = safeScanfInt(&panFile);

        if (returned == 0) {
            return 0;
        }
    }

    int length;
    char *x, *y;
    double z;
    Item *ptr, *prev;
    int i;
    char *filename;
    FILE *fileptr;
    switch (panFile) {
    case 1: // ввод из консоли
        printf("Введите длину списка\n");
        length = -1;
        while (length < 1) {
            printf("Введите целое число больше нуля!\n");
            returned = safeScanfInt(&length);
            if (returned == 0) {
                return 0;
            }
        }

        printf("Введите 1-й элемент\n");
        x = readl("Введите адрес\n");
        if (x == NULL || strlen(x) == 0) {
            free(x);
            free(y);
            return 0;
        }

        y = (char*) malloc(12*sizeof(char));
        if (y == NULL) {
            printf("Не найдено свободной памяти\n");
            free(x);
            return 0;
        }
        printf("Введите кадастровый номер\n");
        returned = safeScanfNum(y);
        if (returned == 0) {
            free(x);
            free(y);
```

```
    return 0;
}

printf("Введите площадь\n");
returned = safeScanfDouble(&z);
if (returned == 0) {
    free(x);
    free(y);
    return 0;
}

// добавляю только после полного чтения всех полей элемента, чтобы не фришить по многу раз
ptr = (Item*) malloc(sizeof(Item));
if (ptr == NULL) {
    printf("Не найдено свободной памяти\n");
    free(x);
    free(y);
    return 0;
}
ptr->adr = (char*) malloc((strlen(x)+1) * sizeof(char));
if (ptr->adr == NULL) {
    printf("Не найдено свободной памяти\n");
    free(ptr);
    free(x);
    free(y);
    return 0;
}
strcpy(ptr->adr, x);
free(x);
x = NULL;

ptr->num = (char*) malloc((11+1) * sizeof(char));
if (ptr->num == NULL) {
    printf("Не найдено свободной памяти\n");
    free(ptr->adr);
    free(ptr);
    free(y);
    return 0;
}
strcpy(ptr->num, y);
free(y);
y = NULL;

ptr->sqr = z;
ptr->next = NULL;
list->head = ptr;

prev = list->head; // в цикле будет создаваться новый ptr, поэтому prev уже будет предыдущим
for (i = 1; i < length; i++) {
    printf("Введите %d-й элемент\n", i+1);
    x = readl("Введите адрес без пробелов\n");
    if (x == NULL || strlen(x) == 0) {
            free(x);
            free(y);
            return 0;
    }

    printf("Введите кадастровый номер\n");
    y = (char*)malloc((11+1)*sizeof(char));
    returned = safeScanfNum(y);
    if (returned == 0) {
        free(x);
        free(y);
        return 0;
    }

    printf("Введите площадь\n");
    returned = safeScanfDouble(&z);
    if (returned == 0) {
        free(x);
        free(y);
        return 0;
    }

    ptr = (Item*) malloc(sizeof(Item));
    if (ptr == NULL) {
        printf("Не найдено свободной памяти\n");
        free(x);
        free(y);
```

```
            return 0;
        }

        ptr->adr = (char*) malloc((strlen(x)+1) * sizeof(char));
        if (ptr->adr == NULL) {
            printf("Не найдено свободной памяти\n");
            free(ptr);
            free(x);
            free(y);
            return 0;
        }
        strcpy(ptr->adr, x);
        free(x);
        x = NULL;

        ptr->num = (char*) malloc((11+1) * sizeof(char));
        if (ptr->num == NULL) {
            printf("Не найдено свободной памяти\n");
            free(ptr->adr);
            free(ptr);
            free(y);
            return 0;
        }
        strcpy(ptr->num, y);
        free(y);
        y = NULL;

        ptr->sqr = z;
        ptr->next = NULL;

        prev->next = ptr;
        prev = ptr;
    }
    // x и y уже зафришены
    break;
case 2: // ввод из текстового файла
    filename = readl("Введите название файла для чтения\n");
    if (filename == NULL || strlen(filename) == 0) {
        return 0;
    }
    fileptr = fopen(filename, "r");
    free(filename);
    if (fileptr == NULL) {
        printf("Ошибка при открытии файла!\n");
        return 0;
    }
    fseek(fileptr, 0, SEEK_SET); // этого по идее не надо

    length = -1;
    // для проверки:
    // printf("Длина списка?\n");
    // returned = safeScanfInt(&length);
    returned = safeFileScanfInt(&length, fileptr);
    if (length < 1 || returned == 0) {
        printf("Введены некорректные данные - длина списка\n");
        fclose(fileptr);
        return 0;
    }

    // ввод адреса элемента 0
    returned = readlFile(&x, fileptr);
    if (returned == 0 || strlen(x) == 0) {
        // принт "нет памяти" написан в readl-функции
        free(x);
        fclose(fileptr);
        return 0;
    }

    // ввод кадастрового номера элемента 0
    returned = readlFile(&y, fileptr);
    if (returned == 0 || strlen(y) == 0) {
        free(x);
        free(y);
        fclose(fileptr);
        return 0;
    }
    if (isValidNum(y) < 1) {
        printf("Введены некорректные данные - номер 1-го элемента\n");
        free(x);
```

```
    free(y);
    fclose(fileptr);
    return 0;
}

// ввод площади элемента 0
returned = safeFileScanfDouble(&z, fileptr);
if (returned == -1) {
    free(x);
    free(y);
    fclose(fileptr);
    return 0;
}
if (returned == 0) {
    printf("Введены некорректные данные - площадь 1-го элемента\n");
    free(x);
    free(y);
    fclose(fileptr);
    return 0;
}

// добавляю только после полного чтения всех полей элемента, чтобы не фришить по многу раз
ptr = (Item*) malloc(sizeof(Item));
if (ptr == NULL) {
    printf("Не найдено свободной памяти\n");
    free(x);
    free(y);
    return 0;
}

ptr->adr = (char*) malloc((strlen(x)+1) * sizeof(char));
if (ptr->adr == NULL) {
    printf("Не найдено свободной памяти\n");
    free(ptr);
    free(x);
    free(y);
    return 0;
}
strcpy(ptr->adr, x);
free(x);
x = NULL;

ptr->num = (char*) malloc((11+1) * sizeof(char));
if (ptr->num == NULL) {
    printf("Не найдено свободной памяти\n");
    free(ptr->adr);
    free(ptr);
    free(y);
    return 0;
}
strcpy(ptr->num, y);
free(y);
y = NULL;

ptr->sqr = z;
ptr->next = NULL;
list->head = ptr;

prev = list->head; // в цикле будет создаваться новый ptr, поэтому prev уже будет предыдущим
for (i = 1; i < length; i++) {
    // ввод адреса элемента i
    returned = readlFile(&x, fileptr);
    if (returned == 0 || strlen(x) == 0) {
        // принт "нет памяти" написан в readl-функции
        free(x);
        // free(y); не надо, т.к он зафришен и заново не аллокнут ещё
        fclose(fileptr);
        return 0;
    }

    // ввод кадастрового номера элемента i
    returned = readlFile(&y, fileptr);
    if (returned == 0 || strlen(y) == 0) {
        free(x);
        free(y);
        fclose(fileptr);
        return 0;
    }
    if (isValidNum(y) < 1) {
```

```
            printf("Введены некорректные данные - номер %d-го элемента\n", i+1);
            free(x);
            free(y);
            fclose(fileptr);
            return 0;
        }

        // ввод площади элемента i
        returned = safeFileScanfDouble(&z, fileptr);
        if (returned == -1) {
            free(x);
            free(y);
            fclose(fileptr);
            return 0;
        }
        if (returned == 0) {
            printf("Введены некорректные данные - площадь %d-го элемента\n", i+1);
            free(x);
            free(y);
            fclose(fileptr);
            return 0;
        }

        ptr = (Item*) malloc(sizeof(Item));
        if (ptr == NULL) {
            printf("Не найдено свободной памяти\n");
            free(x);
            free(y);
            return 0;
        }

        ptr->adr = (char*) malloc((strlen(x)+1) * sizeof(char));
        if (ptr->adr == NULL) {
            printf("Не найдено свободной памяти\n");
            free(ptr);
            free(x);
            free(y);
            return 0;
        }
        strcpy(ptr->adr, x);
        free(x);
        x = NULL;

        ptr->num = (char*) malloc((11+1) * sizeof(char));
        if (ptr->num == NULL) {
            printf("Не найдено свободной памяти\n");
            free(ptr->adr);
            free(ptr);
            free(y);
            return 0;
        }
        strcpy(ptr->num, y);
        free(y);
        y = NULL;

        ptr->sqr = z;
        ptr->next = NULL;

        prev->next = ptr;
        prev = ptr;
    }
    // x и y уже зафришены
    fclose(fileptr);
    break;
case 3: // ввод из бинарного файла
    filename = readl("Введите название файла для чтения\n");
    if (filename == NULL || strlen(filename) == 0) {
        return 0;
    }
    fileptr = fopen(filename, "r");
    free(filename);
    if (fileptr == NULL) {
        printf("Ошибка при открытии файла!\n");
        return 0;
    }
    fseek(fileptr, 0, SEEK_SET); // этого по идее не надо

    length = -1;
    // для проверки:
```

```
// printf("Длина списка?\n");
// returned = safeScanfInt(&length);
returned = safeBinaryScanfInt(&length, fileptr);
if (length < 1 || returned == 0) {
    printf("Введены некорректные данные - длина списка\n");
    fclose(fileptr);
    return 0;
}

// ввод адреса элемента 0
returned = readlBinary(&x, fileptr);
if (returned == 0 || strlen(x) == 0) {
    // принт написан в readl-функции
    free(x);
    fclose(fileptr);
    return 0;
}

// ввод кадастрового номера элемента 0
returned = readlBinary(&y, fileptr);
if (returned == 0 || strlen(y) == 0) {
    free(x);
    free(y);
    fclose(fileptr);
    return 0;
}
if (isValidNum(y) < 1) {
    printf("Введены некорректные данные - номер 1-го элемента\n");
    free(x);
    free(y);
    fclose(fileptr);
    return 0;
}

// ввод площади элемента 0
returned = safeBinaryScanfDouble(&z, fileptr);
if (returned == -1) {
    free(x);
    free(y);
    fclose(fileptr);
    return 0;
}
if (returned == 0) {
    printf("Введены некорректные данные - площадь 1-го элемента\n");
    free(x);
    free(y);
    fclose(fileptr);
    return 0;
}

// добавляю только после полного чтения всех полей элемента, чтобы не фришить ptr по многу раз
ptr = (Item*) malloc(sizeof(Item));
if (ptr == NULL) {
    printf("Не найдено свободной памяти\n");
    free(x);
    free(y);
    return 0;
}

ptr->adr = (char*) malloc((strlen(x)+1) * sizeof(char));
if (ptr->adr == NULL) {
    printf("Не найдено свободной памяти\n");
    free(ptr);
    free(x);
    free(y);
    return 0;
}
strcpy(ptr->adr, x);
free(x);
x = NULL;

ptr->num = (char*) malloc((11+1) * sizeof(char));
if (ptr->num == NULL) {
    printf("Не найдено свободной памяти\n");
    free(ptr->adr);
    free(ptr);
    free(y);
    return 0;
}
```

```
strcpy(ptr->num, y);
free(y);
y = NULL;
ptr->sqr = z;
ptr->next = NULL;

list->head = ptr;

prev = list->head; // в цикле будет создаваться новый ptr, поэтому prev уже будет предыдущим
for (i = 1; i < length; i++) {
   // ввод адреса элемента i
   returned = readlBinary(&x, fileptr);
   if (returned == 0 || strlen(x) == 0) {
      // принт написан в readl-функции
      free(x);
      // free(y); не надо, т.к он зафришен и заново не аллокнут ещё
      fclose(fileptr);
      return 0;
   }

   // ввод кадастрового номера элемента i
   returned = readlBinary(&y, fileptr);
   if (returned == 0 || strlen(y) == 0) {
      free(x);
      free(y);
      fclose(fileptr);
      return 0;
   }
   if (isValidNum(y) < 1) {
      printf("Введены некорректные данные - номер %d-го элемента\n", i+1);
      free(x);
      free(y);
      fclose(fileptr);
      return 0;
   }

   // ввод площади элемента i
   returned = safeBinaryScanfDouble(&z, fileptr);
   if (returned == -1) {
      free(x);
      free(y);
      fclose(fileptr);
      return 0;
   }
   if (returned == 0) {
      printf("Введены некорректные данные - площадь %d-го элемента\n", i+1);
      free(x);
      free(y);
      fclose(fileptr);
      return 0;
   }

   ptr = (Item*) malloc(sizeof(Item));
   if (ptr == NULL) {
      printf("Не найдено свободной памяти\n");
      free(x);
      free(y);
      return 0;
   }

   ptr->adr = (char*) malloc((strlen(x)+1) * sizeof(char));
   if (ptr->adr == NULL) {
      printf("Не найдено свободной памяти\n");
      free(ptr);
      free(x);
      free(y);
      return 0;
   }
   strcpy(ptr->adr, x);
   free(x);

   ptr->num = (char*) malloc((11+1) * sizeof(char));
   if (ptr->num == NULL) {
      printf("Не найдено свободной памяти\n");
      free(ptr->adr);
      free(ptr);
      free(y);
      return 0;
   }
```

```
            strcpy(ptr->num, y);
            free(y);
            ptr->sqr = z;
            ptr->next = NULL;

            prev->next = ptr;
            prev = ptr;
        }
        // x и y уже зафришены
        fclose(fileptr);
        break;
    }

    return 1; // всё ок
}

int outputList (List *list)
{
    Item *ptr = list->head;
    if (ptr == NULL) {
        printf("Список пуст!\n");
        return 1;
    }

    printf("Выберите, куда осуществить вывод:\n\
        (1) в стандартный поток вывода («на экран»)\n\
        (2) в текстовый файл\n\
        (3) в бинарный файл\n");
    int panFile = 0;
    int returned;
    while (panFile < 1  ||  panFile > 3) {
        printf("Введите число от 1 до 3!\n");
        returned = safeScanfInt(&panFile);
        if (returned == 0) {
            return 0;
        }
    }

    char *filename;
    FILE *fileptr;
    switch (panFile) {
    case 1: // вывод в консоль
        printf("Вывод списка, в каждой строке новый элемент\n");
        while (ptr != NULL) {
            printf("%s   %s   %lf\n", ptr->adr, ptr->num, ptr->sqr);
            ptr = ptr->next;
        }
        break;

    case 2: // вывод в текстовый файл
        filename = readl("Введите название файла для записи\n");
        if (filename == NULL || strlen(filename) == 0) {
            return 0;
        }
        fileptr = fopen(filename, "w");
        free(filename);
        if (fileptr == NULL) {
            printf("Ошибка при открытии файла!\n");
            return 0;
        }
        fseek(fileptr, 0, SEEK_SET); // этого по идее не надо
        printf("Вывод списка, в каждой строке новый элемент\n");
        while (ptr != NULL) {
            fprintf(fileptr, "%s %s %lf\n", ptr->adr, ptr->num, ptr->sqr);
            ptr = ptr->next;
        }
        fclose(fileptr);
        break;

    case 3: // вывод в бинарный файл
        filename = readl("Введите название файла для записи\n");
        if (filename == NULL || strlen(filename) == 0) {
            return 0;
        }
        fileptr = fopen(filename, "w");
        free(filename);
        if (fileptr == NULL) {
            printf("Ошибка при открытии файла!\n");
            return 0;
```

```
        }
        fseek(fileptr, 0, SEEK_SET); // этого по идее не надо
        printf("Вывод списка, в каждой строке новый элемент\n");
        while (ptr != NULL) {
            fwrite(ptr->adr, sizeof(char), strlen(ptr->adr), fileptr);
            fwrite(&" ", sizeof(char), 1, fileptr);

            fwrite(ptr->num, sizeof(char), 11, fileptr);
            fwrite(&" ", sizeof(char), 1, fileptr);

            fwrite(&(ptr->sqr), sizeof(double), 1, fileptr);
            fwrite(&"\n", sizeof(char), 1, fileptr);
            ptr = ptr->next;
        }
        fclose(fileptr);
        break;
    }

    return 1;
}

int lenList (List *list)
{
    int len = 0;
    Item *ptr = list->head;
    while (ptr != NULL) {
        len++;
        ptr = ptr->next;
    }

    return len;
}

void swap (Item *prev1, Item *prev2)
{
    Item *temp;
    Item *ptr1 = prev1->next;
    Item *ptr2 = prev2->next;

    temp = ptr2->next;
    ptr2->next = ptr1->next;
    ptr1->next = temp;
    prev1->next = ptr2;
    prev2->next = ptr1;
}

void swapNeghbours (Item *prev1)
{
    Item *ptr1 = prev1->next;
    Item *ptr2 = ptr1->next;

    ptr1->next = ptr2->next;
    ptr2->next = ptr1;
    prev1->next = ptr2;
}

void swapHead (List *list, Item *prev2)
{
    Item *temp;
    Item *ptr1 = list->head;
    Item *ptr2 = prev2->next;

    temp = ptr2->next;
    ptr2->next = ptr1->next;
    ptr1->next = temp;
    list->head = ptr2;
    prev2->next = ptr1;
}

void swapHeadNeghbours (List *list)
{
    Item *ptr1 = list->head;
    Item *ptr2 = ptr1->next;

    ptr1->next = ptr2->next;
    ptr2->next = ptr1;
    list->head = ptr2;
}
```

```
void ptrToIndx (List *list, Item **ptr, int indx)
{
   *ptr = list->head;
   int i;
   for (i = 0; i < indx; i++) {
      *ptr = (*ptr)->next;
   }
}

int compareList (Item *item1, Item *item2, int atr, int dir)
{
   int comp;
   switch (atr) {
      case 1: // адрес
         comp = strcmp(item1->adr, item2->adr);
         break;

      case 2: // номер
         comp = strcmp(item1->num, item2->num);
         break;

      case 3: // площадь
         comp = item1->sqr - item2->sqr;
         break;
   }
   // comp < 0 => знак <, т.е возрастание. dir = 1 - возрастание.
   if ((comp < 0 && dir == 2) || (comp > 0 && dir == 1)) {
      return 1;
   }
   return 0;
}

void combSort (List *list, int atr, int dir)
{
   double factor = 1.25;
   int len = lenList(list);

   int step = len - 1;
   int i;
   Item *ptr1, *prev1, *prev2; // ptr1 нужен, т.к на 1 шаге for'a всегда prev1=NULL
   while (step >= 1) {
      prev1 = NULL;
      ptr1 = list->head;
      ptrToIndx(list, &prev2, step-1);
      for (i = 0; i+step < len; i++) {
         if (compareList(ptr1, prev2->next, atr, dir)) {
            if (step == 1) { // если элементы соседние, то swap работает иначе
               if (prev1 == NULL) {
                  swapHeadNeghbours(list);
               }
               else {
                  swapNeghbours(prev1);
               }
            }
            else {
               if (prev1 == NULL) {
                  swapHead(list, prev2);
               }
               else {
                  swap(prev1, prev2);
               }
            }
         }
         if (prev1 == NULL)
            {prev1 = list->head;}
         else
            {prev1 = prev1->next;}
         ptr1 = prev1->next;
         if (step == 1) {prev2 = ptr1;} // потому что если они рядом, то при swap prev2 уедет, но если они рядом, то prev2=ptr1 -
КОСТЫЛЬ!!!
         // легче было бы просто использовать на каждом шаге фора ptrToIndx, но это долго ведь
         else {prev2 = prev2->next;}
      }//for
      step /= factor;
   }//while
}
// правда, если дерево сверху вниз, то наоборот heapRecUp, просто корень же сверху, ну..
void heapRecDown (List *list, Item *prev1, Item *prev2, int indx, int atr, int dir)
{
```

```
    // if (indx == 0) {
    //     return;
    // } // не нужно, т.к. отсюда приходим только из 1 или 2, которые рассмотрены
    if (indx == 1) {
        if (compareList(list->head, list->head->next, atr, dir)) {
            swapHeadNeghbours(list);
        }
        return;
    }
    if (indx == 2) {
        if (compareList(list->head, list->head->next->next, atr, dir)) {
            swapHead(list, list->head->next); // надо дать prev, так что даю ->next, а не ->next->next
        }
        return;
    }

    if (compareList(prev1->next, prev2->next, atr, dir)) {
        swap(prev1, prev2);

        indx = (indx - 1) / 2; // новый индекс на ptr2
        prev2 = prev1;
        ptrToIndx(list, &prev1, (indx-1)/2 - 1);

        heapRecDown(list, prev1, prev2, indx, atr, dir);
    }
}

void heapBuild (List *list, int atr, int dir)
{
    int len = lenList(list);
    Item *prev1, *prev2;
    int i, j=0; // i - индекс last эл., j - его предок
    prev1 = NULL;
    prev2 = list->head;
    for (i = 1; i < len; i++) { // от i=1, т.к пирамида из 1 эл. - всегда пир. В.
        if (i == 2) {
            // т.к. при i=1 после swap уезжает как раз prev2
            // в других случаях prev2 не в той же ветке, по которой идёт heapRecDown и свапает
            prev2 = list->head->next;
        }
        if (j == 1) {
            prev1 = list->head; // т.к. он тоже мог уехать
        }
        if (j < (i-1)/2) {
            j++;
            if (prev1 == NULL) {
                prev1 = list->head;
            }
            else {
                prev1 = prev1->next;
            }
        }
        heapRecDown(list, prev1, prev2, i, atr, 3 - dir); // 3 - dir, т.к. пирамида строится с монот. наоборот
        prev2 = prev2->next;
    }
}
// правда, если дерево сверху вниз, то наоборот heapRecDown, просто корень же сверху, ну..
void heapRecUp (List *list, Item *prevPar, Item *parent, int indx, int last, int atr, int dir)
{
    if (last == 0) {
        // потому что уже не с кем сравнивать
        // хотя можно было просто в heapSort() вызов RecUp поставить в другое место, и это не нужно - там написал
        return;
    }
    if (indx == 0) {
        Item *ptr1 = list->head->next;
        Item *ptr2 = ptr1->next;
        if (last == 1) {
            ptr2 = NULL; // потому что ptr2 уже вышел из кучи, но ptr1 - ещё нет
        }

        if (compareList(list->head, ptr1, atr, dir)) {
            if (ptr2 != NULL && compareList(ptr1, ptr2, atr, dir)) {
                swapHead(list, ptr1); // т.к нужен prev
                heapRecUp(list, list->head->next, list->head->next->next, 2, last, atr, dir);
            }
            else {
                swapHeadNeghbours(list);
```

```
            heapRecUp(list, list->head, list->head->next, 1, last, atr, dir);
        }
    }
    else if (ptr2 != NULL && compareList(list->head, ptr2, atr, dir)) {
        swapHead(list, ptr1); // т.к нужен prev
        heapRecUp(list, list->head->next, list->head->next->next, 2, last, atr, dir);
    }
    return;
}

int len = lenList(list);
if (indx > len/2) {
    return; // потому что у parent нету листьев
}

if (2*indx+1 > last) {
    return; // т.к тогда ptr1 - элемент, уже исключённый из кучи
}

Item *prev1, *ptr1, *ptr2;
ptrToIndx(list, &prev1, 2*indx+1 - 1);
ptr1 = prev1->next;
ptr2 = ptr1->next;

if (2*indx+2 > last) {
    ptr2 = NULL; // т.к тогда ptr2 - элемент, уже исключённый из кучи, но ptr1 - нет
}

if (compareList(parent, ptr1, atr, dir)) {
    if (ptr2 != NULL && compareList(ptr1, ptr2, atr, dir)) {
        swap(prevPar, ptr1); // ptr1, т.к нужен prev
        heapRecUp(list, ptr1, ptr1->next, 2*indx+2, last, atr, dir);
    }
    else {
        swap(prevPar, prev1);
        heapRecUp(list, prev1, prev1->next, 2*indx+1, last, atr, dir);
    }
}
else if (ptr2 != NULL && compareList(parent, ptr2, atr, dir)) {
    swap(prevPar, ptr1); // т.к нужен prev
    heapRecUp(list, ptr1, ptr1->next, 2*indx+2, last, atr, dir);
}
}

void heapSort (List *list, int atr, int dir)
{
    heapBuild(list, atr, dir);

    int len = lenList(list);
    int i;
    // цикл до 0, т.к при i=0 уже единственный элемент из кучи исключается, и менять ничего не надо
    for (i = len-1; i > 0; i--) { // здесь вылезает n^2, которого можно избежать, если бы List был двусвяз.
        Item *prev2;
        ptrToIndx(list, &prev2, i-1); // если двусвяз., не пришлось бы эту строку каждый раз

        if (i > 1) {
            swapHead(list, prev2); // меняем корень с последним
        }
        else {
            swapHeadNeghbours(list);
        }
        // а можно было это в ифку пихнуть, т.к после else так и так нечего менять
        heapRecUp(list, NULL, NULL, 0, i-1, atr, 3 - dir);
    }
}

int partition (List *list, int left, int right, int atr, int dir)
{
    int pivot = left;
    int i;
    Item *prevPiv, *prevI, *ptrLeft;
    ptrToIndx(list, &ptrLeft, left);
    if (left == 0) {
        prevPiv = NULL;
        prevI = list->head;
    }
    else {
        ptrToIndx(list, &prevPiv, left-1);
        prevI = prevPiv->next;
```

```
    }
    for (i = left+1; i < right+1; i++) {
        if (compareList(ptrLeft, prevI->next, atr, dir)) {
            pivot++; // т.е первый pivot=left+1
            if (prevPiv == NULL) {prevPiv = list->head;}
            else {prevPiv = prevPiv->next;}

            if (i == pivot) { // никогда нет i<pivot, т.к они оба от left+1, а i бежит каждую итерацию, в отличие от pivot
                prevI = prevI->next;
                continue; // т.к зачем свапать с самим собой
            }
            // i > pivot
            if (i - pivot == 1) {
                swapNeghbours(prevPiv);
                continue; // т.к prevI свапнулся, то его не надо двигать
            }
            swap(prevPiv, prevI);
        }
        prevI = prevI->next;
    }

    if (pivot == left) {
        return pivot; // если pivot не изменился, то свапать ничего и не надо, ведь эта часть уже была отсортирована
    }
    if (left == 0) {
        if (list->head == prevPiv) {
            swapHeadNeghbours(list);
        }
        else {
            swapHead(list, prevPiv);
        }
    }
    else {
        Item *prevLeft;
        ptrToIndx(list, &prevLeft, left - 1);
        if (prevLeft->next == prevPiv) {
            swapNeghbours(prevLeft);
        }
        else {
            swap(prevLeft, prevPiv);
        }
    }
    return pivot;
}

void quickSort (List *list, int left, int right, int atr, int dir)
{
    if (left >= right) {
        return;
    }

    int mid = partition(list, left, right, atr, dir);
    quickSort(list, left, mid-1, atr, dir);
    quickSort(list, mid+1, right, atr, dir);
}
```

Листинг 4: Исходный код программы 2 – файл main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "other.h"

int main()
{
    List *list = NULL;
    int panel, len, amm;
    int fl=1, returned, i;
    int pS1, pS2, pS3;
    clock_t start, end;
    double time_used;
    while (fl == 1) {
        printf("Выберите одну из опций:\n\
            (1) Генерация списка\n\
            (2) Вывод списка\n\
            (3) Сортировка списка\n\
            (4) Генерация + сортировка множества списков\n\
            (5) Завершение программы\n");
```

```
panel = 0;
while (panel < 1  ||  panel > 5) {
   printf("Введите число от 1 до 5!\n");
   returned = safeScanfInt(&panel);
   if (returned == 0) {
      fl = 0;
      endOfProgram(list);
      return 0;
   }
}

switch(panel) {
case 1: // генерация списка
   printf("Введите длину списка\n");
   len = 0;
   while (len < 1) {
      printf("Введите число большее нуля!\n");
      returned = safeScanfInt(&len);
      if (returned == 0) {
         fl = 0;
         endOfProgram(list);
         return 0;
      }
   }

   returned = initializeList(&list, len);
   if (returned == 0) {
      fl = 0;
      endOfProgram(list);
      return 0;
   }

   printf("Список создан\n");
   outputList(list);
   break;

case 2: // вывод списка
   outputList(list);
   break;

case 3: // сортировка списка
   returned = sortingList(list);
   if (returned == 0) {
      fl = 0;
      endOfProgram(list);
      return 0;
   }

   printf("Список отсортирован\n");
   outputList(list);
   break;

case 4: // генерация + сортировка списков
   printf("Задайте длину списков - целое число >0\n\
      или 0, чтобы создавать списки (одинаковой) рандомной длины\n");
   len = -1;
   while (len < 0) {
      printf("Введите число большее -1!\n");
      returned = safeScanfInt(&len);
      if (returned == 0) {
         fl = 0;
         endOfProgram(list);
         return 0;
      }
   }

   printf("Задайте количество списков - целое число >0\n\
      или 0, чтобы создать рандомное количество списков\n");
   amm = -1;
   while (amm < 0) {
      printf("Введите число большее -1!\n");
      returned = safeScanfInt(&amm);
      if (returned == 0) {
         fl = 0;
         endOfProgram(list);
         return 0;
      }
   }
   if (amm == 0) {
```

```
    srand(clock());
    amm = rand() % 1000 + 1;
}

printf("Выберите алгоритм сортировки:\n\
        (1) Сортировка расчёской (Comb sort)\n\
        (2) Пирамидальная сортировка (Heap sort)\n\
        (3) Быстрая сортировка (qsort)\n");
pS1 = -1;
while (pS1 < 1 || pS1 > 3) {
    printf("Введите целое число от 1 до 3!\n");
    returned = safeScanfInt(&pS1);
    if (returned == 0) {
        return 0;
    }
}

printf("Выберите поле структуры, по которому будем сортировать:\n\
    (1) Адрес\n\
    (2) Кадастровый номер\n\
    (3) Площадь\n");
pS2 = -1;
while (pS2 < 1 || pS2 > 3) {
    printf("Введите целое число от 1 до 3!\n");
    returned = safeScanfInt(&pS2);
    if (returned == 0) {
        return 0;
    }
}

printf("Выберите направление сортировки (возрастание/убывание:\n\
    (1) Возрастание\n\
    (2) Убывание\n");
pS3 = -1;
while (pS3 < 1 || pS3 > 2) {
    printf("Введите целое число от 1 до 2!\n");
    returned = safeScanfInt(&pS3);
    if (returned == 0) {
        return 0;
    }
}

time_used = 0;
for (i = 0; i < amm; i++) {
    returned = initializeList(&list, len);
    if (returned == 0) {
        fl = 0;
        endOfProgram(list);
        return 0;
    }

    switch(pS1) {
    case 1:
        start = clock();
        combSort(list, pS2, pS3);
        end = clock();
        break;
    case 2:
        start = clock();
        heapSort(list, pS2, pS3);
        end = clock();
        break;
    case 3:
        int lenlist = lenList(list);
        start = clock();
        quickSort(list, 0, lenlist-1, pS2, pS3);
        end = clock();
        break;
    }
    time_used += (double)(end - start) / CLOCKS_PER_SEC;
}
printf("На сортировку %d массивов затрачено %lf секунд\n", amm, time_used);
if (len > 0 && amm > 1) {
    printf("В среднем - %lf секунд\n", time_used / (double)amm);
}

break;
```

```
        case 5: // завершение программы
            fl = 0;
            endOfProgram(list);
            return 0;
        }
    }

    // перебирает все элементы и каждый free(), включая поля элемента, а затем и сам list
    endOfProgram(list); // с принтом "Завершение программы\n"
    return 0;
}
```

Листинг 5: Исходный код программы 2 – файл main.c

```
#ifndef OTHER_H
#define OTHER_H

typedef struct Item {
    char *adr, *num;
    double sqr;
    struct Item *next;
} Item;

typedef struct List {
    Item *head;
} List;

int safeScanfInt (int *target);

int initializeList (List **list, int length);
void outputList (List *list);
int lenList (List *list);
void ptrToIndx (List *list, Item **ptr, int indx);

int sortingList (List *list);
void combSort (List *list, int atr, int dir);
void heapRecDown (List *list, Item *prev1, Item *prev2, int indx, int atr, int dir);
void heapRecUp (List *list, Item *prevPar, Item *parent, int indx, int last, int atr, int dir);
void heapBuild (List *list, int atr, int dir);
void heapSort (List *list, int atr, int dir);
void quickSort (List *list, int left, int right, int atr, int dir);
int partition (List *list, int left, int right, int atr, int dir);
int compareList (Item *item1, Item *item2, int atr, int dir);
void swap (Item *prev1, Item *prev2);
void swapNeghbours (Item *prev1);
void swapHead (List *list, Item *prev2);
void swapHeadNeghbours (List *list);

void freeList (List *list);
void endOfProgram (List *list);

#endif
```

Листинг 4: Исходный код программы 2 – файл main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "other.h"

void freeList(List *list)
{
    if (list == NULL) {
        return;
    }
    Item *ptr = list->head;
    Item *next;
    while (ptr != NULL) {
            next = ptr->next;
            free(ptr->adr);
            free(ptr->num);
            free(ptr);
            ptr = next;
    }

    free(list);
}

void endOfProgram (List *list)
{
```

```
    freeList(list);
    printf("Завершение программы\n");
}

int safeScanfInt (int *target)
{
   int guard;
   int flag = 1;
   while (flag == 1) {
      guard = scanf("%d",target);
      scanf("%*[^\n]");
      scanf("%*c");
      if (guard == EOF) {
         return 0;
      }
      if (guard < 1) {
         printf("Введите целое число!\n");
         continue;
      }
      flag = 0;
   }
   return 1;
}



int initializeList (List **list, int length)
{
   // т.е пользователь вместо ввода длины попросил её сгенерировать
   if (length == 0) {
      srand(clock());
      length = (rand() % 1000) + 1;
   }
   if (list != NULL) {
      freeList(*list);
   }
   *list = (List*) malloc(length*sizeof(List));
   if (*list == NULL) {
      printf("Не найдено свободной памяти\n");
      return 0;
   }
   Item *ptr, *prev;
   int adrlen;
   srand(clock());
   for (int i = 0; i < length; i++) {
      ptr = (Item*)malloc(sizeof(Item));
      if (ptr == NULL) {
         printf("Не найдено свободной памяти\n");
         return 0;
      }
      adrlen = rand() % 100 + 10;
      ptr->adr = (char*)malloc((adrlen + 1) * sizeof(char));
      if (ptr->adr == NULL) {
         printf("Не найдено свободной памяти\n");
         return 0;
      }
      ptr->adr[adrlen] = '\0';
      for (int j = 0; j < adrlen; j++) {
         ptr->adr[j] = ('A' + (rand() % 26)) + (32 * (rand() % 2));
      }
      ptr->num = (char*)malloc((11+1) * sizeof(char));
      if (ptr->num == NULL) {
         printf("Не найдено свободной памяти\n");
         return 0;
      }
      ptr->num[11] = '\0';
      for (int j = 0; j < 11; j++) {
         if (j == 2 || j == 5 || j == 8) {
            ptr->num[j] = ':';
            continue;
         }
         if (j == 3 || j == 4) {
            ptr->num[j] = ('A' + (rand() % 26)) + (32 * (rand() % 2));
            continue;
         }
         if (j == 6 || j == 7) {
            ptr->num[j] = '0';
            continue;
         }
```

```
            ptr->num[j] = '0' + rand() % 10;
        }
        ptr->sqr = (double)rand() / RAND_MAX * 1000 + 0.5;
        ptr->next = NULL;
        if (i == 0) {
            (*list)->head = ptr;
            prev = ptr;
        }
        else {
            prev->next = ptr;
            prev = ptr;
        }
    }

    return 1; // всё ок
}

void outputList (List *list)
{
    Item *ptr = list->head;
    if (ptr == NULL) {
        printf("Список пуст!\n");
        return;
    }

    printf("Вывод последнего созданного списка, в каждой строке новый элемент\n");
    while (ptr != NULL) {
        printf("%s   %s   %lf\n", ptr->adr, ptr->num, ptr->sqr);
        ptr = ptr->next;
    }
}

int lenList (List *list)
{
    int len = 0;
    Item *ptr = list->head;
    while (ptr != NULL) {
        len++;
        ptr = ptr->next;
    }

    return len;
}

void swap (Item *prev1, Item *prev2)
{
    Item *temp;
    Item *ptr1 = prev1->next;
    Item *ptr2 = prev2->next;

    temp = ptr2->next;
    ptr2->next = ptr1->next;
    ptr1->next = temp;
    prev1->next = ptr2;
    prev2->next = ptr1;
}

void swapNeghbours (Item *prev1)
{
    Item *ptr1 = prev1->next;
    Item *ptr2 = ptr1->next;

    ptr1->next = ptr2->next;
    ptr2->next = ptr1;
    prev1->next = ptr2;
}

void swapHead (List *list, Item *prev2)
{
    Item *temp;
    Item *ptr1 = list->head;
    Item *ptr2 = prev2->next;

    temp = ptr2->next;
    ptr2->next = ptr1->next;
    ptr1->next = temp;
    list->head = ptr2;
    prev2->next = ptr1;
}
```

```
void swapHeadNeghbours (List *list)
{
   Item *ptr1 = list->head;
   Item *ptr2 = ptr1->next;

   ptr1->next = ptr2->next;
   ptr2->next = ptr1;
   list->head = ptr2;
}

void ptrToIndx (List *list, Item **ptr, int indx)
{
   *ptr = list->head;
   int i;
   for (i = 0; i < indx; i++) {
      *ptr = (*ptr)->next;
   }
}

int compareList (Item *item1, Item *item2, int atr, int dir)
{
   int comp;
   switch (atr) {
      case 1: // адрес
         comp = strcmp(item1->adr, item2->adr);
         break;

      case 2: // номер
         comp = strcmp(item1->num, item2->num);
         break;

      case 3: // площадь
         comp = item1->sqr - item2->sqr;
         break;
   }
   // comp < 0 => знак <, т.е возрастание. dir = 1 - возрастание.
   if ((comp < 0 && dir == 2) || (comp > 0 && dir == 1)) {
      return 1;
   }
   return 0;
}

void combSort (List *list, int atr, int dir)
{
   double factor = 1.25;
   int len = lenList(list);

   int step = len - 1;
   int i;
   Item *ptr1, *prev1, *prev2;
   while (step >= 1) {
      prev1 = NULL;
      ptr1 = list->head;
      ptrToIndx(list, &prev2, step-1);
      for (i = 0; i+step < len; i++) {
         if (compareList(ptr1, prev2->next, atr, dir)) {
            if (step == 1) { // если элементы соседние, то swap работает иначе
               if (prev1 == NULL) {
                  swapHeadNeghbours(list);
               }
               else {
                  swapNeghbours(prev1);
               }
            }
            else {
               if (prev1 == NULL) {
                  swapHead(list, prev2);
               }
               else {
                  swap(prev1, prev2);
               }
            }
         }
         if (prev1 == NULL)
            {prev1 = list->head;}
         else
            {prev1 = prev1->next;}
         ptr1 = prev1->next;
```

```
            if (step == 1) {prev2 = ptr1;} // потому что если они рядом, то при swap prev2 уедет, но если они рядом, то prev2=ptr1 - КОСТЫЛЬ!!!
            // легче было бы просто использовать на каждом шаге фора ptrToIndx, но это долго ведь
            else {prev2 = prev2->next;}
        }//for
        step /= factor;
    }//while
}
// правда, если дерево сверху вниз, то наоборот heapRecUp, просто корень же сверху, ну..
void heapRecDown (List *list, Item *prev1, Item *prev2, int indx, int atr, int dir)
{
    // if (indx == 0) {
    //     return;
    // } // не нужно, т.к. отсюда приходим только из 1 или 2, которые рассмотрены
    if (indx == 1) {
        if (compareList(list->head, list->head->next, atr, dir)) {
            swapHeadNeghbours(list);
        }
        return;
    }
    if (indx == 2) {
        if (compareList(list->head, list->head->next->next, atr, dir)) {
            swapHead(list, list->head->next); // надо дать prev, так что даю ->next, а не ->next->next
        }
        return;
    }

    if (compareList(prev1->next, prev2->next, atr, dir)) {
        swap(prev1, prev2);

        indx = (indx - 1) / 2; // новый индекс на ptr2
        prev2 = prev1;
        ptrToIndx(list, &prev1, (indx-1)/2 - 1);

        heapRecDown(list, prev1, prev2, indx, atr, dir);
    }
}

void heapBuild (List *list, int atr, int dir)
{
    int len = lenList(list);
    Item *prev1, *prev2;
    int i, j=0; // i - индекс last эл., j - его предок
    prev1 = NULL;
    prev2 = list->head;
    for (i = 1; i < len; i++) { // от i=1, т.к пирамида из 1 эл. - всегда пир. В.
        if (i == 2) {
            prev2 = list->head->next; // т.к. при i=1 после swap уезжает как раз prev2 - КОСТЫЛЬ!!!, как и в combsort
        }
        if (j == 1) {
            prev1 = list->head; // т.к. он тоже мог уехать - костыль
        }
        if (j < (i-1)/2) {
            j++;
            if (prev1 == NULL) {
                prev1 = list->head;
            }
            else {
                prev1 = prev1->next;
            }
        }
        heapRecDown(list, prev1, prev2, i, atr, 3 - dir); // 3 - dir, т.к. пирамида строится с монот. наоборот
        prev2 = prev2->next;
    }
}
// правда, если дерево сверху вниз, то наоборот heapRecDown, просто корень же сверху, ну..
void heapRecUp (List *list, Item *prevPar, Item *parent, int indx, int last, int atr, int dir)
{
    if (last == 0) {
        return; // потому что уже не с кем сравнивать
    }
    if (indx == 0) {
        Item *ptr1 = list->head->next;
        Item *ptr2 = ptr1->next;
        if (last == 1) {
            ptr2 = NULL; // потому что ptr2 уже вышел из кучи, но ptr1 - ещё нет
        }

        if (compareList(list->head, ptr1, atr, dir)) {
```

```
            if (ptr2 != NULL && compareList(ptr1, ptr2, atr, dir)) {
                swapHead(list, ptr1); // т.к нужен prev
                heapRecUp(list, list->head->next, list->head->next->next, 2, last, atr, dir);
            }
            else {
                swapHeadNeghbours(list);
                heapRecUp(list, list->head, list->head->next, 1, last, atr, dir);
            }
        }
        else if (ptr2 != NULL && compareList(list->head, ptr2, atr, dir)) {
            swapHead(list, ptr1); // т.к нужен prev
            heapRecUp(list, list->head->next, list->head->next->next, 2, last, atr, dir);
        }
        return;
    }

    int len = lenList(list);
    if (indx > len/2) {
        return; // потому что у parent нету листьев
    }

    if (2*indx+1 > last) {
        return; // т.к тогда ptr1 - элемент, уже исключённый из кучи
    }

    Item *prev1, *ptr1, *ptr2;
    ptrToIndx(list, &prev1, 2*indx+1 - 1);
    ptr1 = prev1->next;
    ptr2 = ptr1->next;

    if (2*indx+2 > last) {
        ptr2 = NULL; // т.к тогда ptr2 - элемент, уже исключённый из кучи, но ptr1 - нет
    }

    if (compareList(parent, ptr1, atr, dir)) {
        if (ptr2 != NULL && compareList(ptr1, ptr2, atr, dir)) {
            swap(prevPar, ptr1); // ptr1, т.к нужен prev
            heapRecUp(list, ptr1, ptr1->next, 2*indx+2, last, atr, dir);
        }
        else {
            swap(prevPar, prev1);
            heapRecUp(list, prev1, prev1->next, 2*indx+1, last, atr, dir);
        }
    }
    else if (ptr2 != NULL && compareList(parent, ptr2, atr, dir)) {
        swap(prevPar, ptr1); // т.к нужен prev
        heapRecUp(list, ptr1, ptr1->next, 2*indx+2, last, atr, dir);
    }
}

void heapSort (List *list, int atr, int dir)
{
    heapBuild(list, atr, dir);

    int len = lenList(list);
    int i;
    // цикл до 0, т.к при i=0 уже единственный элемент из кучи исключается, и менять ничего не надо
    for (i = len-1; i > 0; i--) { // здесь вылезает n^2, которого можно избежать, если бы List был двусвяз.
        Item *prev2;
        ptrToIndx(list, &prev2, i-1); // если двусвяз., не пришлось бы эту строку каждый раз

        if (i > 1) {
            swapHead(list, prev2); // меняем корень с последним
        }
        else {
            swapHeadNeghbours(list);
        }
        heapRecUp(list, NULL, NULL, 0, i-1, atr, 3 - dir);
    }
}

int partition (List *list, int left, int right, int atr, int dir)
{
    int pivot = left;
    int i;
    Item *prevPiv, *prevI, *ptrLeft;
    ptrToIndx(list, &ptrLeft, left);
    if (left == 0) {
        prevPiv = NULL;
```

```
        prevI = list->head;
    }
    else {
        ptrToIndx(list, &prevPiv, left-1);
        prevI = prevPiv->next;
    }
    for (i = left+1; i < right+1; i++) {
        if (compareList(ptrLeft, prevI->next, atr, dir)) {
            pivot++; // т.е первый pivot=left+1
            if (prevPiv == NULL) {prevPiv = list->head;}
            else {prevPiv = prevPiv->next;}

            if (i == pivot) { // никогда нет i<pivot, т.к они оба от left+1, а i бежит каждую итерацию, в отличие от pivot
                prevI = prevI->next;
                continue; // т.к зачем свапать с самим собой
            }
            // i > pivot
            if (i - pivot == 1) {
                swapNeghbours(prevPiv);
                continue; // т.к prevI свапнулся, то его не надо двигать
            }
            swap(prevPiv, prevI);
        }
        prevI = prevI->next;
    }

    if (pivot == left) {
        return pivot; // если pivot не изменился, то свапать ничего и не надо, ведь эта часть уже была отсортирована
    }
    if (left == 0) {
        if (list->head == prevPiv) {
            swapHeadNeghbours(list);
        }
        else {
            swapHead(list, prevPiv);
        }
    }
    else {
        Item *prevLeft;
        ptrToIndx(list, &prevLeft, left - 1);
        if (prevLeft->next == prevPiv) {
            swapNeghbours(prevLeft);
        }
        else {
            swap(prevLeft, prevPiv);
        }
    }
    return pivot;
}

void quickSort (List *list, int left, int right, int atr, int dir)
{
    if (left >= right) {
        return;
    }

    int mid = partition(list, left, right, atr, dir);
    quickSort(list, left, mid-1, atr, dir);
    quickSort(list, mid+1, right, atr, dir);
}

int sortingList (List *list)
{
    int returned;
    printf("Выберите алгоритм сортировки:\n\
            (1) Сортировка расчёской (Comb sort)\n\
            (2) Пирамидальная сортировка (Heap sort)\n\
            (3) Быстрая сортировка (qsort)\n");
    int pS1 = -1; // pS - panelSort
    while (pS1 < 1 || pS1 > 3) {
        printf("Введите целое число от 1 до 3!\n");
        returned = safeScanfInt(&pS1);
        if (returned == 0) {
            return 0;
        }
    }

    printf("Выберите поле структуры, по которому будем сортировать:\n\
        (1) Адрес\n\
```

```
        (2) Кадастровый номер\n\
        (3) Площадь\n");
    int pS2 = -1;
    while (pS2 < 1 || pS2 > 3) {
        printf("Введите целое число от 1 до 3!\n");
        returned = safeScanfInt(&pS2);
        if (returned == 0) {
            return 0;
        }
    }

    printf("Выберите направление сортировки (возрастание/убывание:\n\
        (1) Возрастание\n\
        (2) Убывание\n");
    int pS3 = -1;
    while (pS3 < 1 || pS3 > 2) {
        printf("Введите целое число от 1 до 2!\n");
        returned = safeScanfInt(&pS3);
        if (returned == 0) {
            return 0;
        }
    }

    switch(pS1) {
    case 1:
        combSort(list, pS2, pS3);
        break;
    case 2:
        heapSort(list, pS2, pS3);
        break;
    case 3:
        int lenlist = lenList(list);
        quickSort(list, 0, lenlist-1, pS2, pS3);
        break;
    }
    return 1;
}
```

5. Тестовые примеры

Таблица 1: Тестовые примеры программы 1

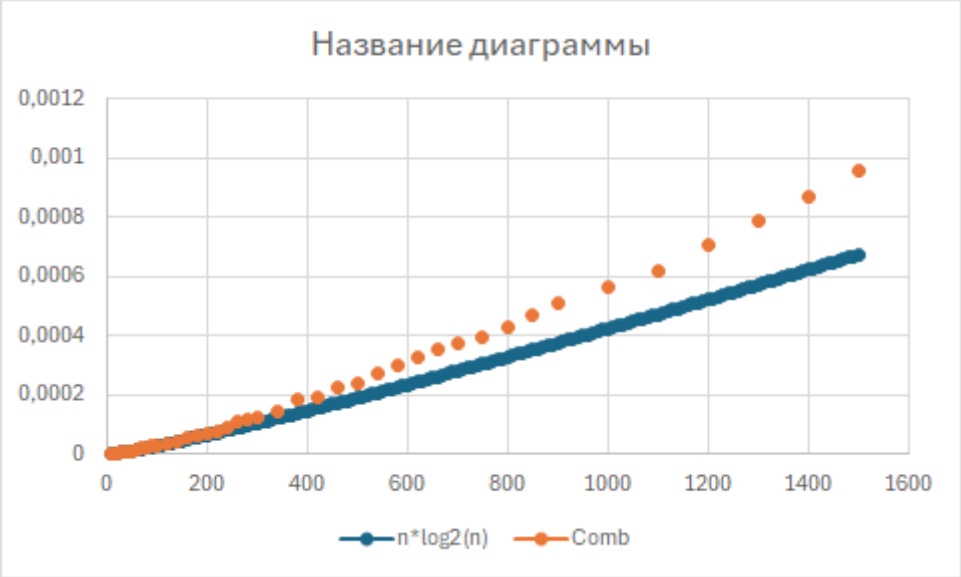
Ввод	Ожидаемый результат	Полученный результат
0	Ошибка	Ошибка
1 - Ввод	Выбор источника	Выбор источника
2 - TXT	Ожидание имени файла	Ожидание имени файла
in	Ошибка	Ошибка
1 - Ввод	Выбор источника	Выбор источника
2 - TXT	Ожидание имени файла	Ожидание имени файла
in.txt	Данные загружены	Данные загружены
2 - Вывод	Выбор источника	Выбор источника
1 - Экран	Данные выведены	Данные выведены
3 - Сортировка	Выбор метода	Выбор метода
1 - Comb	Выбор поля	Выбор поля
1 - Адрес	Выбор направления	Выбор направления
1 - Возрастание	Данные отсортированы	Данные отсортированы
2 - Вывод	Выбор источника	Выбор источника
3 - BIN	Ожидание имени файла	Ожидание имени файла
out.txt	Данные выгружены	Данные выгружены
1 - Ввод	Выбор источника	Выбор источника
3 - BIN	Ожидание имени файла	Ожидание имени файла
out.txt	Данные загружены	Данные загружены
3 - Сортировка	Выбор метода	Выбор метода
3 - Quick	Выбор поля	Выбор поля
3 - Площадь	Выбор направления	Выбор направления
2 - Убывание	Данные отсортированы	Данные отсортированы

6. Результаты таймирования

Таблица 2: Тестовые примеры программы 2 с результатами таймирования
N – длина списка, Comb – сортировка расчёской, Heap – пирамидальная, Quick - быстрая

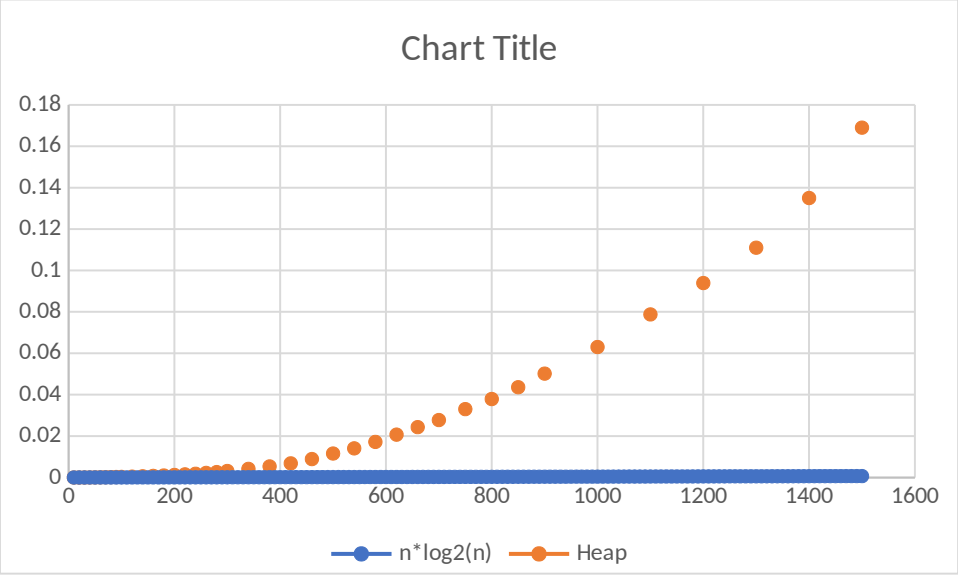
N	Comb	Heap	Quick
10	0.000002	0.000003	0.000001
20	0.000005	0.000008	0.000003
30	0.000006	0.000020	0.000006
40	0.000009	0.000036	0.000008
50	0.000012	0.000061	0.000014
60	0.000017	0.000094	0.000021
70	0.000021	0.000130	0.000026
80	0.000024	0.000174	0.000033
90	0.000027	0.000211	0.000041
100	0.000032	0.000279	0.000054
120	0.000038	0.000419	0.000074
140	0.000041	0.000583	0.000105
160	0.000056	0.000775	0.000133
180	0.000062	0.000981	0.000160
200	0.000073	0.001229	0.000198
220	0.000078	0.001511	0.000228
240	0.000093	0.001816	0.000279
260	0.000111	0.002201	0.000332
280	0.000116	0.002607	0.000381
300	0.000127	0.003137	0.000428
340	0.000143	0.004160	0.000558
380	0.000182	0.005340	0.000676
420	0.000195	0.006817	0.000837
460	0.000227	0.008896	0.001028
500	0.000241	0.011578	0.001254
540	0.000272	0.014074	0.001484
580	0.000299	0.017189	0.001736
620	0.000328	0.020703	0.002055
660	0.000352	0.024323	0.002432
700	0.000373	0.027748	0.002729
750	0.000398	0.033031	0.003195
800	0.000430	0.037901	0.003743
850	0.000470	0.043604	0.004178
900	0.000511	0.050179	0.004745
1000	0.000564	0.063027	0.005987
1100	0.000617	0.078777	0.007642
1200	0.000704	0.093924	0.009088
1300	0.000787	0.111000	0.011153
1400	0.000869	0.135000	0.013349
1500	0.000957	0.169000	0.015200

График 1: Сравнение сортировки расчёской с nlogn



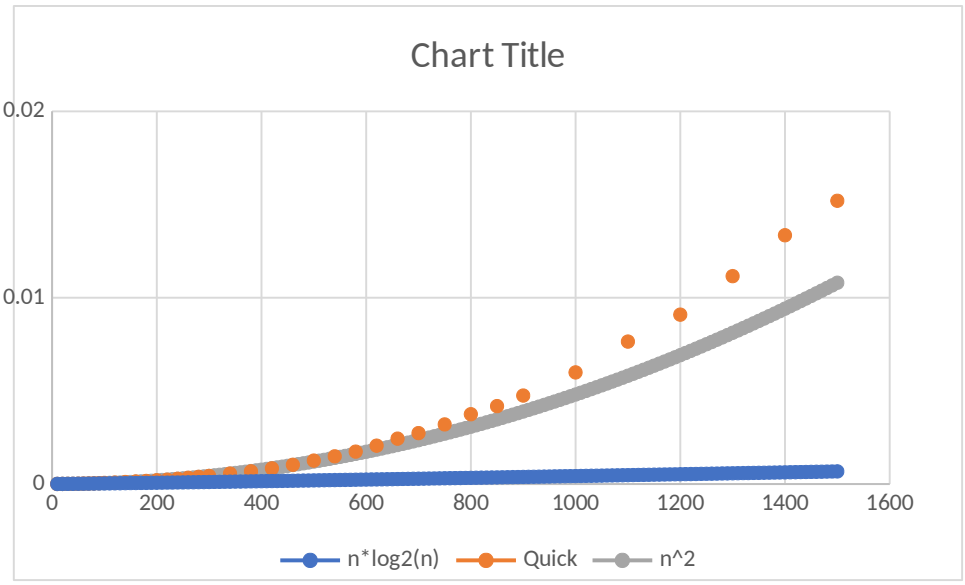
Сортировка расчёской имеет среднюю сложность O(nlogn), и хотя в худшем случае будет O(n^2), при достаточно большой выборке всё же будет похожесть именно на O(nlogn)

График 2: Сравнение heap-сортировки с nlogn



Сортировка пирамидой Вильямса всегда имеет сложность O(nlogn). Однако же из-за того, что список односвязный, алгоритм вырождается в сложность O(n^2 * logn)

График 3: Сравнение быстрой сортировки с nlogn и n^2



Алгоритм быстрой сортировки также имеет сложность O(nlogn), но по тем же причинам вырождается в квадрат.

7. Выводы

В ходе выполнения данной работы на примере программы, обрабатывающей строки символов, были рассмотрены принципы построения программ на языке С, использующих массивы и указатели, структуры данных:

- Разбиение программы на несколько .c – файлов, .h файлы.
- Контроль ошибок в работе с памятью при помощи Valgrind.
- Оперирование \0.
- Динамическое выделение и освобождение памяти, с использованием функций: malloc(), realloc(), free().
- Создание самописных структур данных и их организация в виде массива.
- Таймирование вычислений. В результате замеров и усреднения времени сортировки сгенерированных массивов получены и отображены на графиках данные о быстродействии рассмотренных типов сортировок. Можно сказать, что общие тренды совпадают с предполагаемыми