Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Факультет безопасности информационных технологий

Дисциплина:

«Алгоритмы и структуры данных»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №3

«PSRS-сортировка с очередью на связном списке и поиск методом Бойера–Мура»

Выполнил:
Суханкулиев М.,
студент группы N3246

_____
(подпись)

Проверил:
Ерофеев С. А.

_____
(отметка о выполнении)

_____
(подпись)

Санкт-Петербург

2025 г.

ВВЕДЕНИЕ

Цель работы – Разработать программу PSRS-сортировки, используя обычную очередь на базе связного списка. После сортировки выполнить поиск методом Бойера-Мура.

Для достижения поставленной цели необходимо решить следующие **задачи**:

- Разработать блок-схему алгоритма;
- Составить спецификацию всех переменных;
- Реализовать программу на языке С;
- Провести тестирование программы.

1 АЛГОРИТМ ПРОГРАММЫ

1.1 Описание алгоритма PSRS

Начало.

Шаг 1. Исходные данные считываются из файла во временную очередь `Queue* queue`, реализованную на связном списке. Каждый элемент — указатель на `double`.

Из очереди данные копируются в массив `double* buffer`. Вычисляется общее количество элементов `n`.

Массив `buffer` разделяется на `p` равных подмассивов (учитывается остаток при делении), каждый из которых передаётся потоку для локальной сортировки. Размер подмассива для каждого потока определяется как `base_size = n / p`, остаток — `rem = n % p`.

Шаг 2. В каждом потоке вызывается функция `local_sort`, в которой выполняется быстрая сортировка `qsort` массива `thread_data[i].array`. Результат — `p` отсортированных подмассивов, сохранённых в `sub_arrays[i]`.

Шаг 3. Формируется вспомогательный массив `samples` из `p` элементов каждого отсортированного подмассива по формуле: $j \cdot \left(\frac{n}{p^2}\right)$, где $j = 0,1,2,\dots,p-1$.

Шаг 4. Общий размер массива `samples` будет `p * p = p^2`. Сортируем вспомогательный массив `samples` с помощью быстрой сортировки (`qsort`).

Шаг 5. Формируем массив разделителей (`splitters`) из элементов `samples`, взятых под индексами $k \cdot p + \left[\frac{p}{2}\right] - 1$, где $k = 1,2,\dots,p-1$. Таким образом, `splitters` содержит `p-1` разделитель: $a_1, a_2, \dots, a_{p-1}$.

Шаг 6. На каждом потоке разбиваем локально отсортированные подмассивы `sub_arrays[i]` на `p` групп по значениям разделителей `splitters`, то есть получаем группы: $(-\infty, a_1], (a_1, a_2], \dots, (a_{p-1}, +\infty)$. Элементы каждой группы добавляются в соответствующую очередь `queues[j]` с помощью `enqueue`.

Шаг 7. Для каждой из `p` очередей вызывается `dequeue`, данные из которых объединяются в итоговые массивы `sub_arrays[i]`. Таким образом, каждый поток получает по одному подмассиву, содержащему элементы своей группы от всех потоков.

Шаг 8. Для каждого массива `sub_arrays[i]` запускается функция `merge_sort`, в которой выполняется финальная сортировка методом `qsort`. Это обеспечивает локально отсортированные блоки для каждого потока.

Шаг 9. Все `p` отсортированных подмассивов собираются в один результирующий массив `result`, выполняется последовательная запись.

Шаг 10. Итоговый массив `result` записывается в выходной файл `sorted_result.txt` с помощью `write_to_file`.

Конец.

1.2 Описание алгоритма Бойера-Мура

Начало.

1. Инициализация: В начале загружаются шаблон для поиска и строка для сопоставления. Для удобства работы алгоритм строит две таблицы:

- **Таблица стоп-символов**: эта таблица хранит индексы последних вхождений символов в шаблоне, исключая последний символ. В случае несовпадения символа в строке таблица позволяет сдвигать шаблон на несколько позиций.
- **Таблица сдвигов**: эта таблица помогает учесть совпавшие суффиксы и минимизировать сдвиг при несовпадении, сдвигая шаблон таким образом, чтобы совпавший суффикс снова оказался на своём месте.

2. Поиск совпадений: Алгоритм начинает с последнего символа шаблона и сравнивает его с соответствующим символом строки. Если символы совпадают, процесс продолжается с предыдущих символов шаблона. Это происходит до тех пор, пока либо не будет найдено полное совпадение, либо не произойдёт несовпадение.

3. Обработка несовпадений: В случае несовпадения алгоритм сдвигает шаблон вправо. Для этого используются две эвристики:

- **Эвристика стоп-символа** (`badchar`): при несовпадении символа шаблона с символом строки, шаблон сдвигается на величину, соответствующую разнице между текущей позицией в строке и значением из таблицы стоп-символов для данного символа. Эта операция реализована в коде через поиск индекса последнего вхождения символа в шаблоне (переменная `bad_idx`).
- **Эвристика совпавшего суффикса** (`shift`): если символ шаблона и строки не совпали, но существует совпавший суффикс, алгоритм сдвигает шаблон так, чтобы совпавший суффикс снова оказался на своём месте. Эта эвристика реализована в функции `preprocess_good`, где вычисляются сдвиги, которые минимизируют изменения положения шаблона.

4. Продолжение поиска: После сдвига шаблона процесс продолжается с нового положения шаблона и строки. Если найдено совпадение, алгоритм выводит индекс первого совпадения шаблона. Если в строке больше нет совпадений шаблона, поиск завершается.

5. Выход: Поиск продолжается до тех пор, пока не будут найдены все совпадения шаблона или пока не будет достигнут конец строки. В моем коде, если ввод пользователя некорректен или поиск не дал результата, пользователю выводится сообщение "`Значение не найдено`", и программа продолжает ожидать новый ввод.

Конец.

1.3 Блок-схема алгоритма

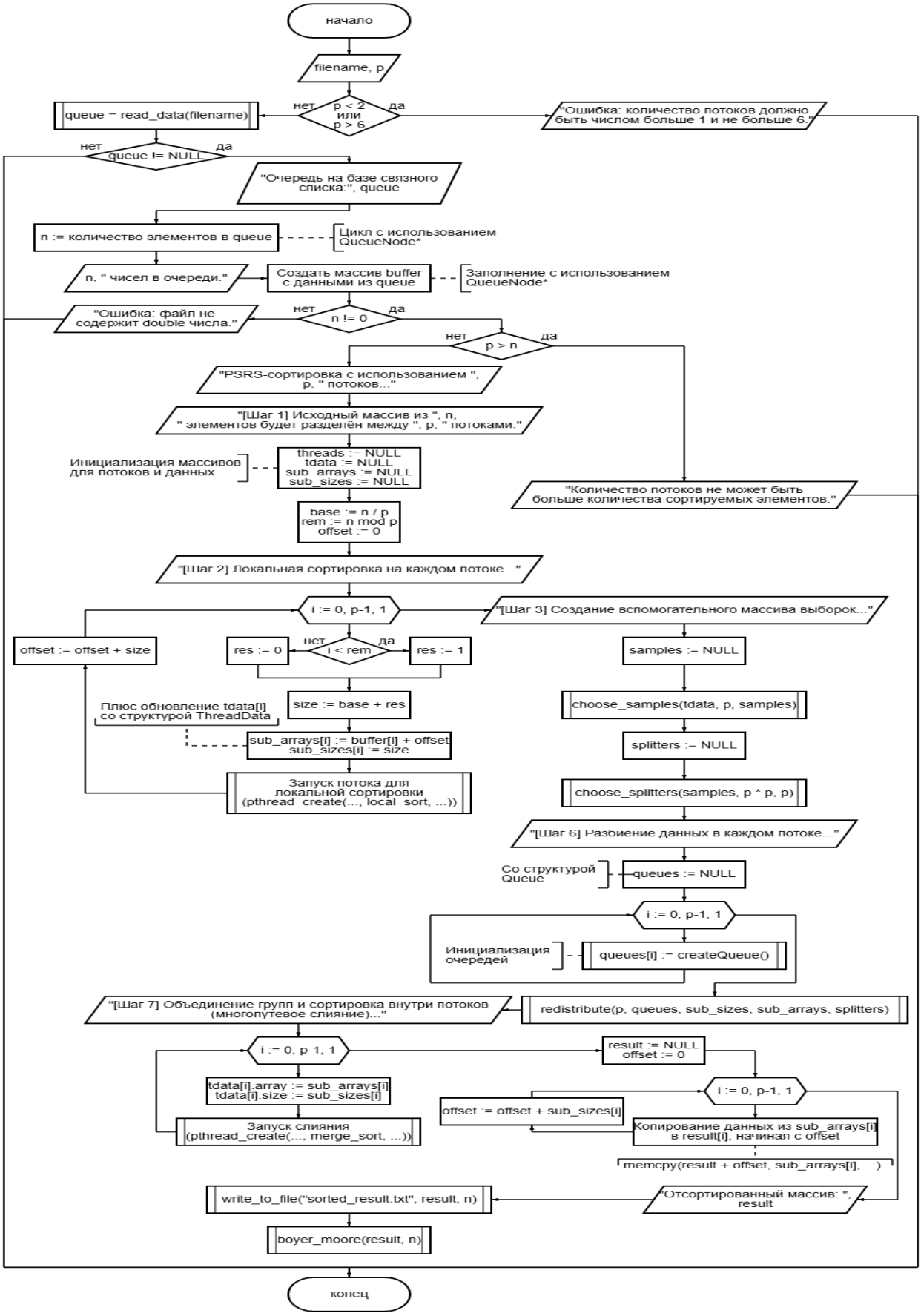

Рисунок 1 – Основная схема алгоритма

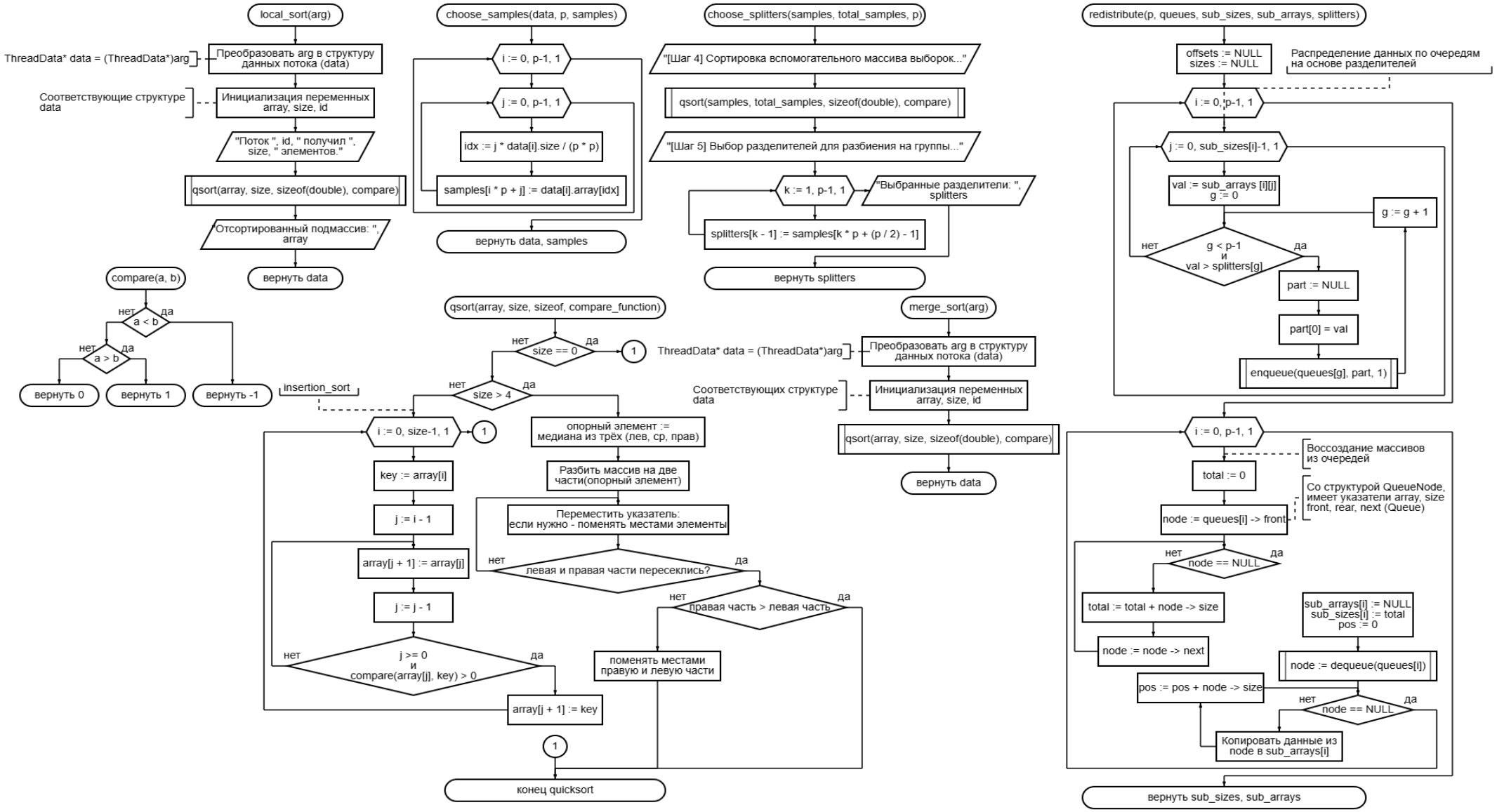

Рисунок 2 – Подпрограммы (psrs)

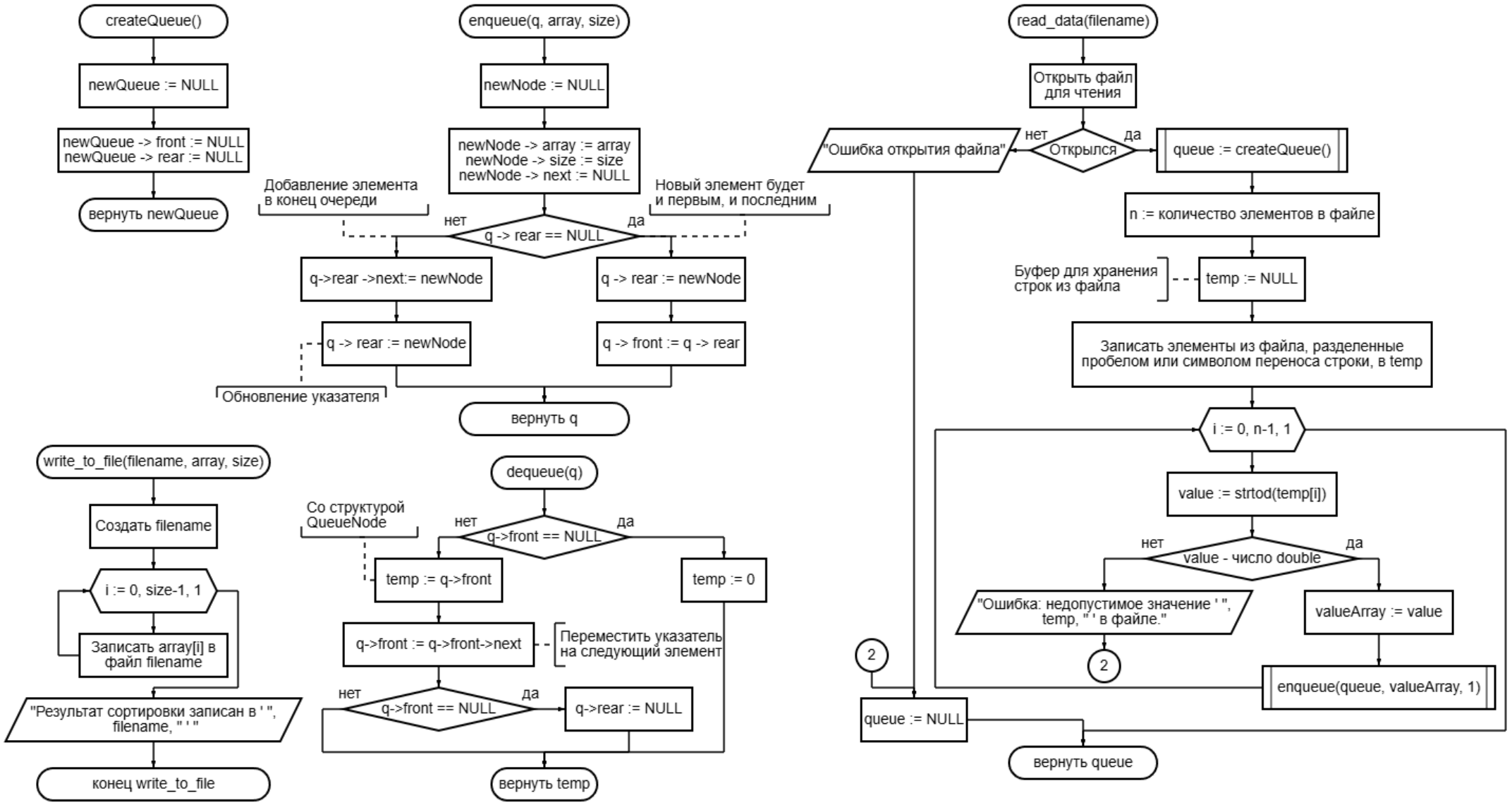

Рисунок 3 – Подпрограммы (queue)

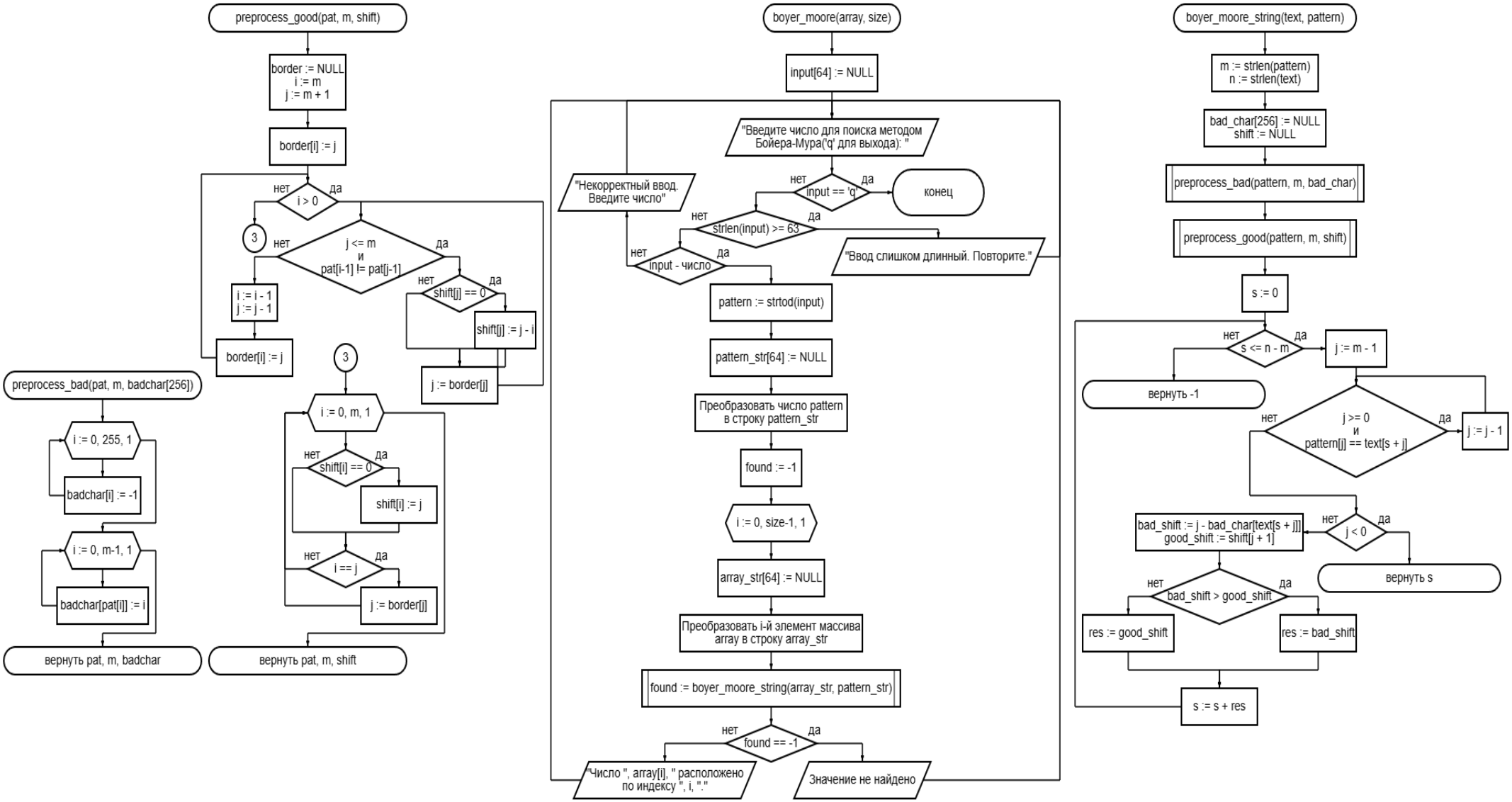

Рисунок 4 – Подпрограммы (Boyer-Moore)

2 СПЕЦИФИКАЦИЯ ПЕРЕМЕННЫХ

Переменная	Описание	Тип использования	Тип	Размер (байт)	Диапазон значений
filename	Путь к файлу, содержащему данные для сортировки	Входная	const char	8	Путь к файлу в файловой системе (строка)
p	Количество потоков	Входная	int	4	[2, 6]
queue	Очередь для хранения чисел	Промежуточная	Queue	8	Указатель на структуру очереди
base	Основание для деления массива на части без остатка	Промежуточная	int	4	[0, 1 073 741 824]
n	Количество чисел в массиве	Промежуточная	int	4	[0, 2 147 483 647]
sig_atomic_t	Флаг для обработки сигнала SIGINT	Промежуточная	volatile	4	[0, 1]
threads	Массив потоков	Промежуточная	pthread_t	8	Системный указатель на поток (зависит от платформы)
tdata	Массив структур с данными для каждого потока	Промежуточная	ThreadData	8	Указатель на структуру с информацией о потоках
samples	Массив выборок для вычисления разделителей	Промежуточная	double	8	[2, 30]
splitters	Массив разделителей для многопутевого слияния	Промежуточная	double	8	[1, 5]
sub_arrays	Массив подмассивов для каждого потока	Промежуточная	double	8	Массив указателей на подмассивы
sub_sizes	Массив размеров подмассивов для каждого потока	Промежуточная	int	4	[0, 2 147 483 647]
current	Указатель на текущий элемент в очереди	Промежуточная	QueueNode	8	Указатель на текущий элемент в очереди
node	Указатель на текущий узел очереди	Промежуточная	QueueNode	8	Указатель на узел очереди
index	Индекс текущего элемента в массиве	Промежуточная	int	4	[0, 2 147 483 646]
buffer	Массив для хранения элементов из очереди	Промежуточная	double	8	[-1.7977e + 308, 1.7977e + 308]
sizes	Размеры подмассивов после разбиения на группы	Промежуточная	int	4	[0, 2 147 483 647]
part	Элемент массива для распределения по очередям	Промежуточная	double	8	[-1.7977e + 308, 1.7977e + 308]
total	Общее количество элементов, распределённых по очередям	Промежуточная	int	4	[0, 2 147 483 647]
offsets	Смещения для данных при перераспределении по потокам	Промежуточная	int	4	[0, 2 147 483 647]
g	Индекс текущей группы при разбиении данных по разделителям	Промежуточная	int	4	[0, 5]
offset	Смещение для подсчета и копирования элементов	Промежуточная	int	4	[0, 2 147 483 647]
rem	Остаток элементов при делении на количество потоков	Промежуточная	int	4	[0, 5]
result	Массив для хранения отсортированных данных	Выходная	double	8	[-1.7977e + 308, 1.7977e + 308]

Примечание: размер переменных в памяти указан для стандартных платформ x86-64.

3 РЕАЛИЗАЦИЯ ПРОГРАММЫ

3.1 Описание программы

Программа написана на языке C и состоит из пяти файлов:

1. **Заголовочный файл (headers.h)** – содержит объявления функций и структур данных, которые используются в реализации программы.
2. **Основной файл (main.c)** – включает функцию `main()`, которая управляет процессом выполнения программы.
3. **Файл с реализациями функций (psrs.c)** – содержит всех функций, реализующих основную логику алгоритма PSRS.
4. **Файл с реализациями очереди (queue.c)** – содержит функции для работы с очередями.
5. **Файл с реализациями алгоритма Бойера-Мура (bm.c)** – содержит функции для поиска значения в отсортированном массиве с использованием полного алгоритма Бойера-Мура.

3.2 Код программы

3.2.1 headers.h

```
#ifndef HEADERS_H
#define HEADERS_H

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <math.h>
#include <ctype.h>
#include <signal.h>

#define MAX_INPUT 64  // Максимальная длина вводимой строки

// Структура данных для потока
typedef struct {
    double* array;
    int size;
    int id;
} ThreadData;

// Узел очереди
typedef struct QueueNode {
    double* array;
    int size;
    struct QueueNode* next;
} QueueNode;

// Очередь на основе связного списка
typedef struct Queue {
    QueueNode* front;  // Указатель на начало
    QueueNode* rear;   // Указатель на конец
} Queue;

// Глобальные переменные
extern double* splitters;
extern double** sub_arrays;
```

```
extern int* sub_sizes;
extern volatile sig_atomic_t exit_flag; // Флаг завершения (SIGINT)

// Очередь: функции создания, работы и очистки
Queue* createQueue();
void enqueue(Queue* q, double* array, int size);
QueueNode* dequeue(Queue* q);
int isEmpty(Queue* q);
void freeQueue(Queue* q);

// Обработчик сигнала SIGINT (Ctrl+C)
void handle_sigint(int sig);

// Функции сортировки и распределения
int compare(const void* a, const void* b);
void* local_sort(void* arg);
void choose_samples(ThreadData* data, int p, double* samples);
void choose_splitters(double* samples, int total_samples, int p);
void redistribute(int p, Queue** queues); // Перераспределение элементов по разделителям
void* merge_sort(void* arg);              // Финальное многопутевое слияние

// Алгоритм Бойера-Мура для поиска строки (значение double преобразуется в строку)
void preprocess_bad(const char* pat, int m, int badchar[256]);  // Эвристика стоп-символа
void preprocess_good(const char* pat, int m, int* shift);       // Эвристика совпавшего суффикса
void boyer_moore(double* array, int size);                      // Обёртка для поиска значения double в
     массиве через Бойера-Мура по строковому представлению
int boyer_moore_string(const char* text, const char* pattern);  // Основной алгоритм Бойера-Мура для
     поиска шаблона в тексте

// Работа с файлами и вводом/выводом
Queue* read_data(const char* filename);
void write_to_file(const char* filename, double* array, int size);

int main(int argc, char* argv[]);

#endif // HEADERS_H
```

3.2.2 psrs.c

```
#include "headers.h"

// Сравнение двух элементов для qsort
int compare(const void* a, const void* b) {
    double da = *(double*)a;
    double db = *(double*)b;
    return (da > db) - (da < db);
}

// Локальная сортировка фрагмента данных
void* local_sort(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    qsort(data->array, data->size, sizeof(double), compare);
    printf("Поток %d получил %d элементов. Отсортированный подмассив:\n", data->id, data->size);
    for (int i = 0; i < data->size; ++i)
        printf("%.15g ", data->array[i]);
    printf("\n");
    return NULL;
}

// Выбор сэмплов: индексы определяются на основе размера данных
void choose_samples(ThreadData* data, int p, double* samples) {
    for (int i = 0; i < p; ++i) {
        for (int j = 0; j < p; ++j) {
            int idx = j * data[i].size / (p * p);
            samples[i * p + j] = data[i].array[idx];  // Запись сэмпла
        }
    }
}

// Выбор разделителей (splitters) для разбиения на группы
void choose_splitters(double* samples, int total_samples, int p) {
    printf("[Шаг 4] Сортировка вспомогательного массива выборок...\n");
    qsort(samples, total_samples, sizeof(double), compare);  // Сортировка сэмплов для выбора
      разделителей
```

```
    printf("[Шаг 5] Выбор разделителей для разбиения на группы...\n");
    for (int k = 1; k < p; ++k)
        splitters[k - 1] = samples[k * p + (p / 2) - 1];     // Используем медиану каждого блока
    printf("Выбранные разделители: ");
    for (int i = 0; i < p - 1; ++i)
        printf("%.15g ", splitters[i]);
    printf("\n");
}

// Распределение данных по очередям на основе выбранных разделителей
void redistribute(int p, Queue** queues) {
    int* offsets = calloc(p * p, sizeof(int));
    int** sizes = malloc(p * sizeof(int*));
    for (int i = 0; i < p; ++i)
        sizes[i] = calloc(p, sizeof(int));  // Инициализация массива размеров для каждого потока
    // Перебор данных и распределение по очередям на основе разделителей
    for (int i = 0; i < p; ++i) {
        for (int j = 0; j < sub_sizes[i]; ++j) {
            double val = sub_arrays[i][j];  // Получаем значение для распределения
            int g = 0;
            while (g < p - 1 && val > splitters[g]) g++; // Определяем группу для значения
            double* part = malloc(sizeof(double));       // Выделяем память для элемента
            part[0] = val;
            enqueue(queues[g], part, 1);  // Добавляем элемент в соответствующую очередь
        }
    }
    // Воссоздание массивов из очередей
    for (int i = 0; i < p; ++i) {
        int total = 0;
        QueueNode* node = queues[i]->front;
        while (node != NULL) {
            total += node->size;
            node = node->next;
        }
        sub_arrays[i] = malloc(total * sizeof(double));
        sub_sizes[i] = total;
        int pos = 0;
        while ((node = dequeue(queues[i])) != NULL) {
            memcpy(sub_arrays[i] + pos, node->array, node->size * sizeof(double));  // Копируем данные
     из очереди в массив
            pos += node->size;
            free(node->array);
            free(node);
        }
    }
    for (int i = 0; i < p; ++i) {
        free(sizes[i]);
    }
    free(sizes);
    free(offsets);
}

// Финальная сортировка после объединения
void* merge_sort(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    qsort(data->array, data->size, sizeof(double), compare);
    return NULL;
}
```

3.2.3 queue.c

```
#include "headers.h"

// Функция для создания новой очереди (инициализация пустой очереди)
Queue* createQueue() {
    Queue* newQueue = (Queue*)malloc(sizeof(Queue));
    newQueue->front = newQueue->rear = NULL;
    return newQueue;
}

// Функция для добавления элемента в очередь (в конец очереди)
void enqueue(Queue* q, double* array, int size) {
    QueueNode* newNode = (QueueNode*)malloc(sizeof(QueueNode));
    newNode->array = array;
```

```
    newNode->size = size;
    newNode->next = NULL;
    if (q->rear == NULL) {
        q->front = q->rear = newNode;
        return;
    }
    q->rear->next = newNode;
    q->rear = newNode;
}

// Функция для извлечения элемента из очереди (удаление первого элемента)
QueueNode* dequeue(Queue* q) {
    if (q->front == NULL)
        return NULL;
    QueueNode* temp = q->front;
    q->front = q->front->next;
    if (q->front == NULL)
        q->rear = NULL;
    return temp;
}

// Функция для проверки, пуста ли очередь
int isEmpty(Queue* q) {
    return q->front == NULL;
}

// Освобождение памяти, занятой очередью и её элементами
void freeQueue(Queue* q) {
    while (!isEmpty(q)) {
        QueueNode* node = dequeue(q);
        free(node->array);
        free(node);
    }
    free(q);
}

// Функция для чтения данных из файла в очередь
Queue* read_data(const char* filename) {
    FILE* f = fopen(filename, "r");
    if (!f) {
        perror("Ошибка открытия файла");
        return NULL;
    }
    Queue* queue = createQueue();
    char temp[256]; // Буфер для чтения строки
    char* endptr;   // Указатель для проверки корректности преобразования строки в число
    while (fscanf(f, "%s", temp) == 1) {
        double value = strtod(temp, &endptr);
        // Проверка: корректный double, нет мусора в строке, значение конечно
        if (endptr == temp || *endptr != '\0' || !isfinite(value)) {
            fprintf(stderr, "Ошибка: недопустимое значение \"%s\" в файле.\n", temp);
            fclose(f);
            return NULL;
        }
        double* valueArray = (double*)malloc(sizeof(double));
        *valueArray = value;
        enqueue(queue, valueArray, 1);
    }
    fclose(f);
    return queue;
}

// Функция для записи данных в файл
void write_to_file(const char* filename, double* array, int size) {
    FILE* f = fopen(filename, "w");
    if (!f) {
        perror("Ошибка создания файла для записи");
        return;
    }
    for (int i = 0; i < size; ++i)
        fprintf(f, "%.15g ", array[i]);
    fprintf(f, "\n");
```

```
    fclose(f);
    printf("Результат сортировки записан в '%s'.\n", filename);
}
```

3.2.4 bm.c

```
#include "headers.h"

// Построение таблицы "плохих символов" для эвристики bad character
// badchar[c] хранит индекс последнего вхождения символа c в шаблоне
void preprocess_bad(const char* pat, int m, int badchar[256]) {
    for (int i = 0; i < 256; i++) {
        badchar[i] = -1;  // по умолчанию символ не встречается
    }
    for (int i = 0; i < m; i++) {
        badchar[(unsigned char)pat[i]] = i;
    }
}

// Построение таблицы сдвигов для эвристики совпавшего суффикса
void preprocess_good(const char* pat, int m, int* shift) {
    int* border = malloc((m + 1) * sizeof(int));
    int i = m, j = m + 1;
    border[i] = j;
    // Этап 1: заполняем таблицу границ (border positions)
    while (i > 0) {
        while (j <= m && pat[i - 1] != pat[j - 1]) {
            if (shift[j] == 0)
                shift[j] = j - i;
            j = border[j];
        }
        border[--i] = --j;
    }
    // Этап 2: заполняем оставшиеся нули в shift[]
    for (i = 0; i <= m; i++) {
        if (shift[i] == 0)
            shift[i] = j;
        if (i == j)
            j = border[j];
    }
    free(border);
}

// Обёртка: ввод числа, преобразование в строку и поиск
void boyer_moore(double* array, int size) {
    char input[MAX_INPUT];
    // Устанавливаем обработчик сигнала SIGINT (Ctrl+C)
    signal(SIGINT, handle_sigint);
    while (1) {
        if (exit_flag) {
            printf("Программа остановлена.\n");
            break;
        }
        printf("\nВведите число для поиска методом Бойера-Мура('q' для выхода): ");
        if (!fgets(input, MAX_INPUT, stdin)) break;
        if (strncmp(input, "q", 1) == 0) break;
        if (strlen(input) >= MAX_INPUT - 1) {
            printf("Ввод слишком длинный. Повторите.\n");
            while (getchar() != '\n');
            continue;
        }
        // Преобразуем ввод в число
        char* endptr;
        double pattern = strtod(input, &endptr);
        if (endptr == input || (*endptr != '\n' && *endptr != '\0')) {
            printf("Некорректный ввод. Введите число.\n");
            continue;
        }
        // Преобразуем число для поиска в строку (точность до 15 значащих цифр)
        char pattern_str[MAX_INPUT];
        snprintf(pattern_str, MAX_INPUT, "%.15g", pattern);
        int found = -1;
        for (int i = 0; i < size; ++i) {
            // Преобразуем i-й элемент массива в строку
```

```
            char array_str[MAX_INPUT];
            snprintf(array_str, MAX_INPUT, "%.15g", array[i]);
            // Ищем шаблон в строковом представлении элемента массива
            found = boyer_moore_string(array_str, pattern_str);
            if (found != -1) {
                // Дополнительно сравниваем числа с учётом машинной точности
                if (fabs(array[i] - pattern) < 1e-9) {
                    printf("Число %.15g расположено по индексу %d.\n", array[i], i);
                    break;
                }
            }
        }
        if (found == -1) {
            printf("Значение не найдено\n");
        }
    }
}

// Основной алгоритм Бойера-Мура для поиска pattern в text
// Возвращает индекс первого вхождения или -1
int boyer_moore_string(const char* text, const char* pattern) {
    int m = strlen(pattern);
    int n = strlen(text);
    int bad_char[256];
    int* shift = calloc(m + 1, sizeof(int));
    preprocess_bad(pattern, m, bad_char);
    preprocess_good(pattern, m, shift);
    int s = 0;  // текущий сдвиг шаблона
    while (s <= n - m) {
        int j = m - 1;
        while (j >= 0 && pattern[j] == text[s + j]) {
            j--;
        }
        if (j < 0) {
            free(shift);
            return s;  // совпадение найдено
        } else {
            int bad_shift = j - bad_char[(unsigned char)text[s + j]];
            int good_shift = shift[j + 1];
            s += (bad_shift > good_shift) ? bad_shift : good_shift;
        }
    }
    free(shift);
    return -1;  // не найдено
}
```

3.2.5 main.c

```
#include "headers.h"

double* splitters;
double** sub_arrays;
int* sub_sizes;
volatile sig_atomic_t exit_flag = 0;

// Обработчик сигнала SIGINT
void handle_sigint(int sig) {
    (void)sig;
    exit_flag = 1;
}

// Основная функция
int main(int argc, char* argv[]) {
    signal(SIGINT, handle_sigint);
    // Проверка корректности аргументов командной строки
    if (argc != 3) {
        printf("Использование: %s <входной_файл> <кол-во_потоков>\n", argv[0]);
        return 1;
    }
    const char* filename = argv[1];
    int p = atoi(argv[2]);
    if (p < 2 || p > 6) {
        printf("Ошибка: количество потоков должно быть числом больше 1 и не больше 6.\n");
        return 1;
```

```
}
// Чтение данных из файла в очередь
Queue* queue = read_data(filename);
if (!queue) {
    return 1;
}
printf("Очередь на базе связного списка:\n");
QueueNode* current = queue->front;
int count = 0;
while (current != NULL) {
    printf("%.15g ", *(current->array));
    current = current->next;
    count++;
}
printf("\n\n");
int n = 0;
QueueNode* node = queue->front;
while (node != NULL) {
    n++;
    node = node->next;
}
// Выделяем память и заполняем массив данными из очереди
double* buffer = malloc(n * sizeof(double));
if (!buffer) {
    fprintf(stderr, "Ошибка выделения памяти для массива.\n");
    freeQueue(queue);
    return 1;
}
int index = 0;
while (!isEmpty(queue)) {
    QueueNode* node = dequeue(queue);
    buffer[index++] = *node->array;
    free(node->array);
    free(node);
}
printf("%d чисел в очереди.\n", n);
// Проверка на корректность данных
if (n == 0) {
    fprintf(stderr, "Ошибка: файл не содержит double числа.\n");
    free(buffer);
    return 1;
}
if (p > n) {
    printf("Количество потоков не может быть больше количества сортируемых элементов\n");
    free(buffer);
    return 1;
}
printf("PSRS-сортировка с использованием %d потоков...\n\n", p);
printf("[Шаг 1] Исходный массив из %d элементов будет разделён между %d потоками.\n", n, p);
// Инициализация массивов для потоков и данных
pthread_t* threads = malloc(p * sizeof(pthread_t));
ThreadData* tdata = malloc(p * sizeof(ThreadData));
sub_arrays = malloc(p * sizeof(double*));
sub_sizes = malloc(p * sizeof(int));
// Локальная сортировка на каждом потоке
printf("[Шаг 2] Локальная сортировка на каждом потоке...\n");
int base = n / p, rem = n % p, offset = 0;
for (int i = 0; i < p; ++i) {
    int size = base + (i < rem);      // Размер подмассива с учётом остатка
    sub_arrays[i] = buffer + offset;  // Указатель на начало подмассива
    sub_sizes[i] = size;
    tdata[i] = (ThreadData){ sub_arrays[i], size, i };
    // Запуск потока для локальной сортировки
    pthread_create(&threads[i], NULL, local_sort, &tdata[i]);
    offset += size;
}
// Ожидание завершения всех потоков
for (int i = 0; i < p; ++i) {
    pthread_join(threads[i], NULL);
}
// Создание вспомогательного массива сэмплов (выборок) из локально отсортированных фрагментов
printf("[Шаг 3] Создание вспомогательного массива сэмплов...\n");
```

```
    double* samples = malloc(p * p * sizeof(double));
    choose_samples(tdata, p, samples);
    splitters = malloc((p - 1) * sizeof(double));
    choose_splitters(samples, p * p, p);
    free(samples);
    // Разбиение данных в каждом потоке согласно выбранным разделителям
    printf("[Шаг 6] Разбиение данных в каждом потоке...\n");
    // Передаем уже созданные очереди в redistribute
    Queue** queues = malloc(p * sizeof(Queue*));
    for (int i = 0; i < p; ++i) {
        queues[i] = createQueue();
    }
    redistribute(p, queues);
    // Многопутевое слияние: запуск потоков для сортировки каждой группы
    printf("[Шаг 7] Объединение групп и сортировка внутри потоков (многопутевое слияние)...\n");
    for (int i = 0; i < p; ++i) {
        tdata[i].array = sub_arrays[i];
        tdata[i].size = sub_sizes[i];
        pthread_create(&threads[i], NULL, merge_sort, &tdata[i]);
    }
    // Ожидание завершения всех потоков слияния
    for (int i = 0; i < p; ++i) {
        pthread_join(threads[i], NULL);
    }
    // Объединение отсортированных подмассивов в итоговый результат
    double* result = malloc(n * sizeof(double));
    offset = 0;
    for (int i = 0; i < p; ++i) {
        memcpy(result + offset, sub_arrays[i], sub_sizes[i] * sizeof(double));
        offset += sub_sizes[i];
    }
    printf("\nОтсортированный массив:\n");
    for (int i = 0; i < n; ++i) {
        printf("%.15g ", result[i]);  // Вывод каждого элемента
    }
    printf("\n\n");
    write_to_file("sorted_result.txt", result, n);
    boyer_moore(result, n);
    // Освобождение всех выделенных ресурсов
    for (int i = 0; i < p; ++i) {
        free(sub_arrays[i]);
    }
    freeQueue(queue);
    free(sub_arrays);
    free(sub_sizes);
    free(splitters);
    free(threads);
    free(tdata);
    free(result);
    free(buffer);
    for (int i = 0; i < p; ++i) {
        free(queues[i]);
    }
    free(queues);
    return 0;
}
```

4 ТЕСТИРОВАНИЕ ПРОГРАММЫ

В процессе тестирования не было обнаружено утечек памяти или других проблем (примеры ниже).

4.1 Скриншоты тестирования

```
Введите число для поиска методом Бойера-Мура('q' для выхода): q
==126151==
==126151== HEAP SUMMARY:
==126151==     in use at exit: 0 bytes in 0 blocks
==126151==   total heap usage: 191 allocs, 191 frees, 17,400 bytes allocated
==126151==
==126151== All heap blocks were freed -- no leaks are possible
==126151==
==126151== For lists of detected and suppressed errors, rerun with: -s
==126151== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Рисунок 5 – Valgrind

```
┌──(kali㉿kali)-[~/Desktop/algos3]
└─$ ./psrs filea.txt 6
Ошибка открытия файла: No such file or directory

┌──(kali㉿kali)-[~/Desktop/algos3]
└─$ ./psrs file.txt 25
Ошибка: количество потоков должно быть числом больше 1 и не больше 6.

┌──(kali㉿kali)-[~/Desktop/algos3]
└─$ ./psrs file.txt -e 25
Использование: ./psrs <входной_файл> <кол-во_потоков>

┌──(kali㉿kali)-[~/Desktop/algos3]
└─$ ./psrs file.txt 6
Очередь на базе связного списка:


0 чисел в очереди.
Ошибка: файл не содержит double числа.

┌──(kali㉿kali)-[~/Desktop/algos3]
└─$ ./psrs file.txt 6
Ошибка: недопустимое значение "asd" в файле.

┌──(kali㉿kali)-[~/Desktop/algos3]
└─$ ./psrs file.txt 6
Ошибка: недопустимое значение "5e500" в файле.
```

Рисунок 6 – Пример 1

```
┌──(kali㉿kali)-[~/Desktop/algos3]
└─$ ./psrs file.txt 2
Очередь на базе связного списка:
1 1.2 5.3 -0.5 1e+300

5 чисел в очереди.
PSRS-сортировка с использованием 2 потоков...

[Шаг 1] Исходный массив из 5 элементов будет разделён между 2 потоками.
[Шаг 2] Локальная сортировка на каждом потоке...
Поток 0 получил 3 элементов. Отсортированный подмассив:
1 1.2 5.3
Поток 1 получил 2 элементов. Отсортированный подмассив:
-0.5 1e+300
[Шаг 3] Создание вспомогательного массива сэмплов...
[Шаг 4] Сортировка вспомогательного массива выборок...
[Шаг 5] Выбор разделителей для разбиения на группы...
Выбранные разделители: 1
[Шаг 6] Разбиение данных в каждом потоке...
[Шаг 7] Объединение групп и сортировка внутри потоков (многопутевое слияние)...

Отсортированный массив:
-0.5 1 1.2 5.3 1e+300

Результат сортировки записан в 'sorted_result.txt'.

Введите число для поиска методом Бойера-Мура('q' для выхода): 1
Число 1 расположено по индексу 1.

Введите число для поиска методом Бойера-Мура('q' для выхода): 1.2
Число 1.2 расположено по индексу 2.

Введите число для поиска методом Бойера-Мура('q' для выхода): 5.3
Число 5.3 расположено по индексу 3.

Введите число для поиска методом Бойера-Мура('q' для выхода): 25
Значение не найдено

Введите число для поиска методом Бойера-Мура('q' для выхода): 0

Введите число для поиска методом Бойера-Мура('q' для выхода): -0.5
Число -0.5 расположено по индексу 0.

Введите число для поиска методом Бойера-Мура('q' для выхода): exit
Некорректный ввод. Введите число.

Введите число для поиска методом Бойера-Мура('q' для выхода): 0.00000000000000000000000000044444444
Значение не найдено

Введите число для поиска методом Бойера-Мура('q' для выхода): 12222222222222222222222222222222222222222222222222222222222222226.00555555555555555555555
55555555555555555555555555555
Ввод слишком длинный. Повторите.

Введите число для поиска методом Бойера-Мура('q' для выхода): q
```

Рисунок 7 – Пример 2

```
┌──(kali㉿kali)-[~/Desktop/algos3]
└─$ ./psrs file.txt 6
Очередь на базе связного списка:
1 1.2 5.3 -0.5 1e+300 450.5454654 1e-300 1e+300 1e-100 1e+100 42 1 -1 3 2 1.23 1000 10000 100000 1000000 10000000 100000000 1000000000 0 0 0 1000 100 99
99 10000000000 1.23e+45 3.14159e-05 1.7976e+308 -1.7976e+308 1e-308

35 чисел в очереди.
PSRS-сортировка с использованием 6 потоков...
[Шаг 1] Исходный массив из 35 элементов будет разделён между 6 потоками.
[Шаг 2] Локальная сортировка на каждом потоке...
Поток 0 получил 6 элементов. Отсортированный подмассив:
-0.5 1 1.2 5.3 450.5454654 1e+300
Поток 3 получил 6 элементов. Отсортированный подмассив:
0 100000 1000000 10000000 100000000 1000000000
Поток 2 получил 6 элементов. Отсортированный подмассив:
-1 1.23 Поток 5 получил 5 элементов. Отсортированный подмассив:
-1.7976e+308 1e-308 3.14159e-05 1.23e+45 1.7976e+308
Поток 4 получил 6 элементов. Отсортированный подмассив:
0 0 100 1000 9999 10000000000
Поток 1 получил 6 элементов. Отсортированный подмассив:
1e-300 1e-100 1 42 1e+100 1e+300
2 3 1000 10000
[Шаг 3] Создание вспомогательного массива сэмплов...
[Шаг 4] Сортировка вспомогательного массива выборок...
[Шаг 5] Выбор разделителей для разбиения на группы...
Выбранные разделители: -1 -0.5 0 0 1e-300
[Шаг 6] Разбиение данных в каждом потоке...
[Шаг 7] Объединение групп и сортировка внутри потоков (многопутевое слияние)...
Отсортированный массив:
-1.7976e+308 -1 -0.5 0 0 0 1e-308 1e-300 1e-100 3.14159e-05 1 1 1.2 1.23 2 3 5.3 42 100 450.5454654 1000 1000 9999 10000 100000 1000000 10000000 1000000
00 1000000000 10000000000 1.23e+45 1e+100 1e+300 1e+300 1.7976e+308

Результат сортировки записан в 'sorted_result.txt'.

Введите число для поиска методом Бойера-Мура('q' для выхода): 1
Число 1 расположено по индексу 10.

Введите число для поиска методом Бойера-Мура('q' для выхода): asd
Некорректный ввод. Введите число.

Введите число для поиска методом Бойера-Мура('q' для выхода): q
```

Рисунок 8 – Пример 3

ЗАКЛЮЧЕНИЕ

В ходе работы была разработана и реализована программа для сортировки данных с использованием алгоритма PSRS, который применяет многопоточность и очередь на базе связного списка для эффективной обработки данных. Также был интегрирован алгоритм поиска значения в отсортированном массиве методом Бойера-Мура с применением эвристик плохого символа и хорошего суффикса.

Программа корректно выполняет сортировку с разделением данных на подмассивы и их последующим объединением, а также предоставляет эффективный механизм поиска, обеспечивающий высокую точность при сравнении числовых значений с учётом машинной точности.

Тестирование показало стабильную работу программы, отсутствие утечек памяти и других ошибок.

Выполненная работа позволила изучить и реализовать алгоритм PSRS на основе связного списка, а также освоить метод поиска Бойера-Мура.