

**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 №4

«PSRS-сортировка с деком на связном списке и записью в красно-чёрное дерево»

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 дек на базе связного списка. Результаты записать в красно-черное дерево.

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

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

1 АЛГОРИТМ PSRS

1.1 Описание алгоритма

Начало.

Шаг 1. Исходные данные считываются из входного файла и записываются в дек Deque* deque, реализованный на базе связного списка. Каждый элемент дека содержит указатель на одно число типа double.

Шаг 2. Из дека данные последовательно копируются в массив buffer. Определяется общее количество элементов n.

Шаг 3. Массив buffer разделяется на p подмассивов (с учётом остатка $n \% p$), каждый из которых передаётся соответствующему потоку.

Шаг 4. В каждом потоке запускается функция local_sort, в которой выполняется быстрая сортировка qsort() для подмассива.

Шаг 5. Из каждого отсортированного подмассива выбирается p сэмплов по индексам $j \cdot (\text{size}/p)$, где $j = 0..p-1$. Общий массив samples содержит p^2 элементов.

Шаг 6. Массив samples сортируется, после чего из него формируется массив splitters — p-1 разделитель, каждый из которых соответствует медиане блока. Разделители выбираются по индексам $k \cdot p + (p/2) - 1$, где $k = 1..p-1$.

Шаг 7. Каждый поток разбивает свой отсортированный подмассив на группы в соответствии с разделителями splitters. Полученные значения добавляются в Deque-структуры buckets[i], каждая из которых соответствует одной группе.

Шаг 8. После завершения группировки каждый Deque преобразуется в обычный массив sub_arrays[i].

Шаг 9. Для каждого потока выполняется финальная сортировка полученного массива методом qsort() (многопутевое слияние).

Шаг 10. Отсортированные части объединяются в результирующий массив result.

Шаг 11. Массив result записывается в выходной файл sorted_result.txt.

Шаг 12. Каждый элемент массива result вставляется в красно-чёрное дерево RBTree.

Шаг 13. Выполняется экспорт дерева в формат DOT и его визуализация с помощью Graphviz.

Шаг 14. Пользователь может ввести значение для поиска в дереве. При нахождении числа отображается его ранг (позиция в отсортированном массиве) и цвет узла.

Конец.

1.2 Описание дека

Дек реализован на базе связного списка с динамическими узлами, каждый из которых хранит массив чисел. Структура поддерживает следующие операции:

- Создание дека и проверка на пустоту.
- Вставка элементов с начала (`insertFront`) и с конца (`insertRear`).
- Удаление элементов с начала (`deleteFront`) и с конца (`deleteRear`).
- Чтение чисел из файла и добавление их в дек.
- Освобождение памяти и запись отсортированных данных в файл.

Каждое число из файла оборачивается в массив из одного элемента и добавляется в конец дека.

1.3 Описание красно-черного дерева

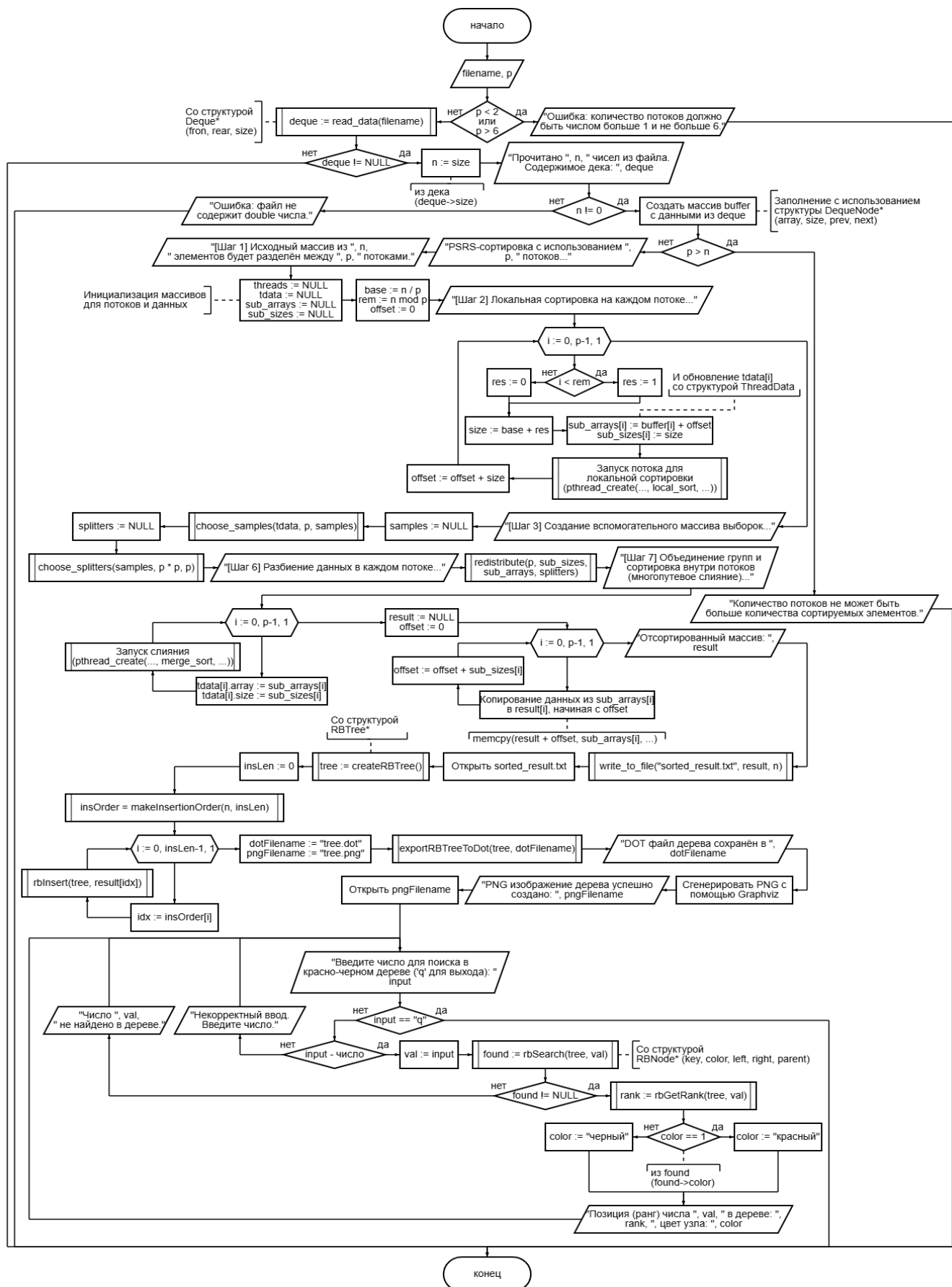
Красно-черное дерево — это самобалансирующееся двоичное дерево поиска, в котором каждый узел содержит ключ и цвет (красный или черный). Дерево обеспечивает логарифмическую сложность вставки и поиска благодаря поддержанию следующих свойств:

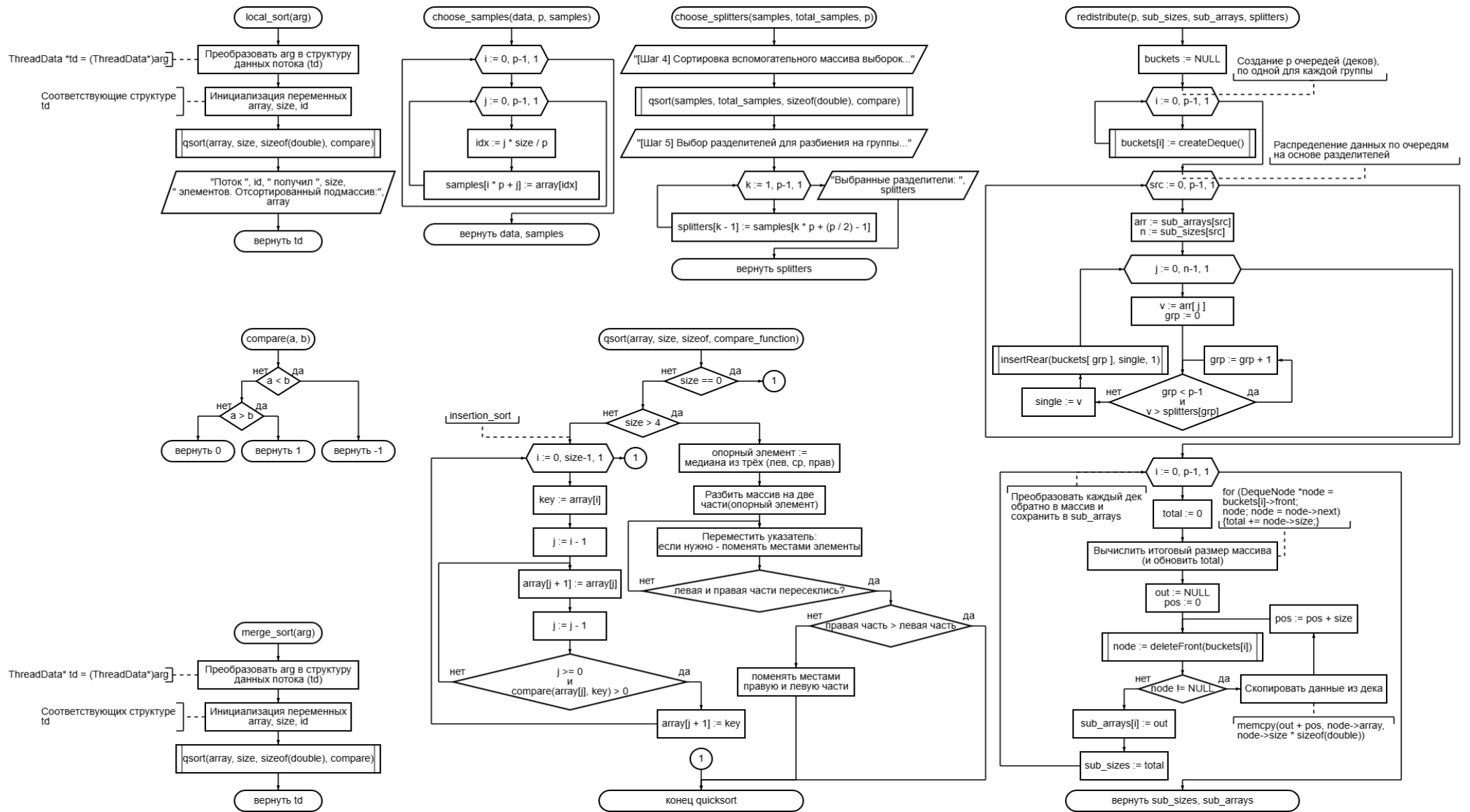
1. Каждый узел промаркирован красным или чёрным цветом
2. Корень и конечные узлы (листья) дерева — чёрные
3. У красного узла родительский узел — чёрный
4. Все простые пути из любого узла x до листьев содержат одинаковое количество чёрных узлов
5. Чёрный узел может иметь чёрного родителя

В данной реализации:

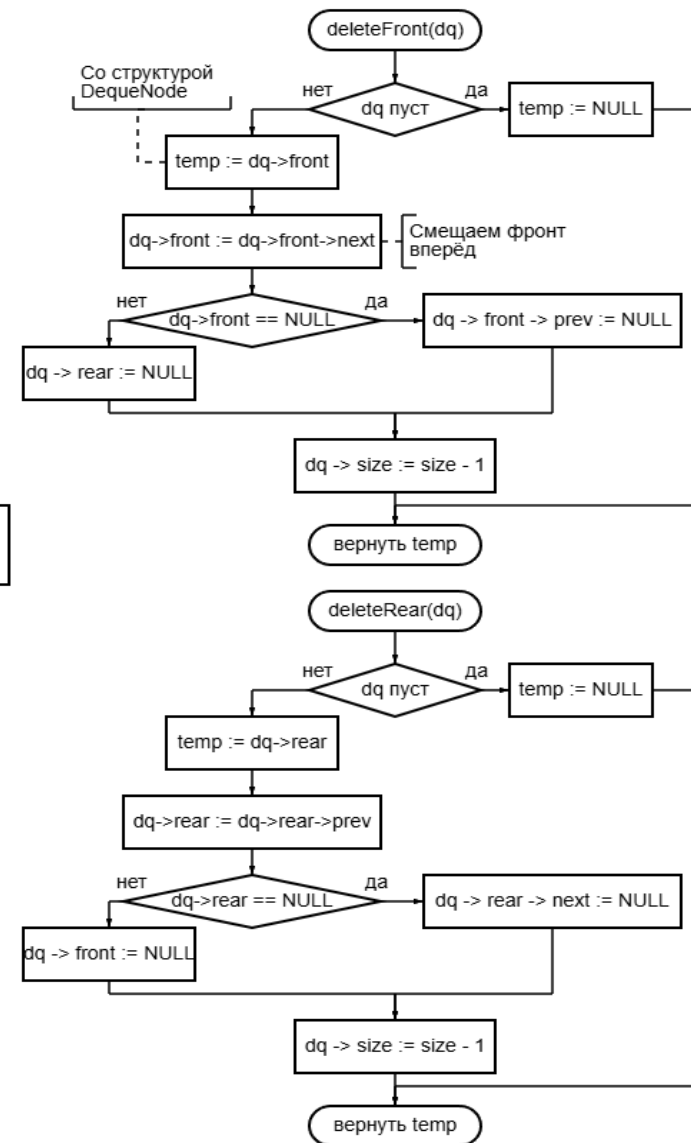
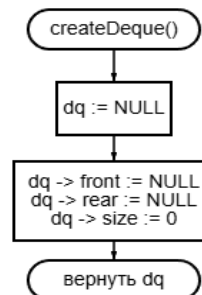
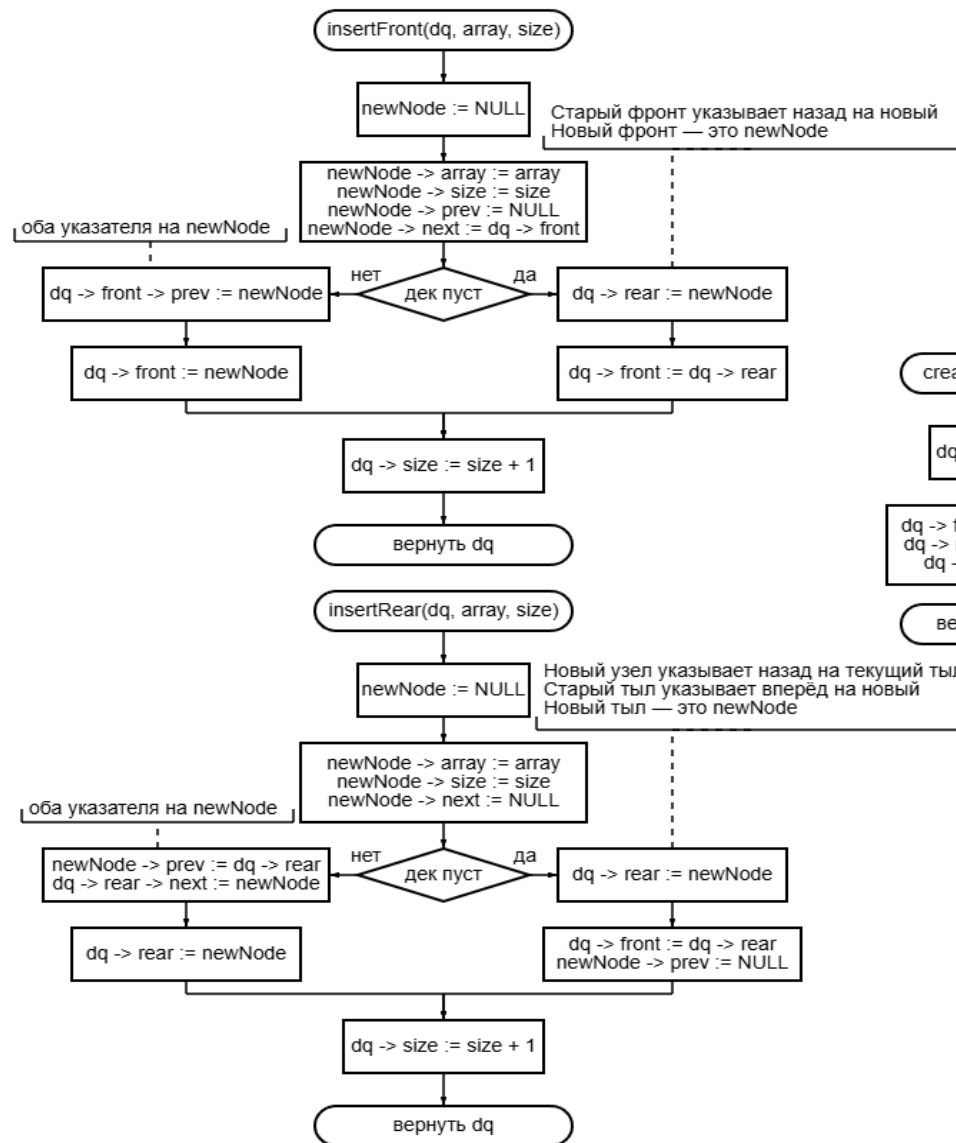
- Функция `createRBTree` и `createNode` создают дерево и новые узлы.
- Вставка `rbInsert` добавляет элемент и вызывает `rbFixInsert`, которая восстанавливает балансировку с помощью поворотов (`leftRotate` и `rightRotate`) и перекраски.
- Поиск осуществляется функцией `rbSearch`.
- `freeRBTree` рекурсивно освобождает память всех узлов.
- `makeInsertionOrder` формирует эффективный порядок вставки для построения сбалансированного дерева на основе массива, разделяя его на подотрезки по медиане.

1.4 Блок-схема алгоритма

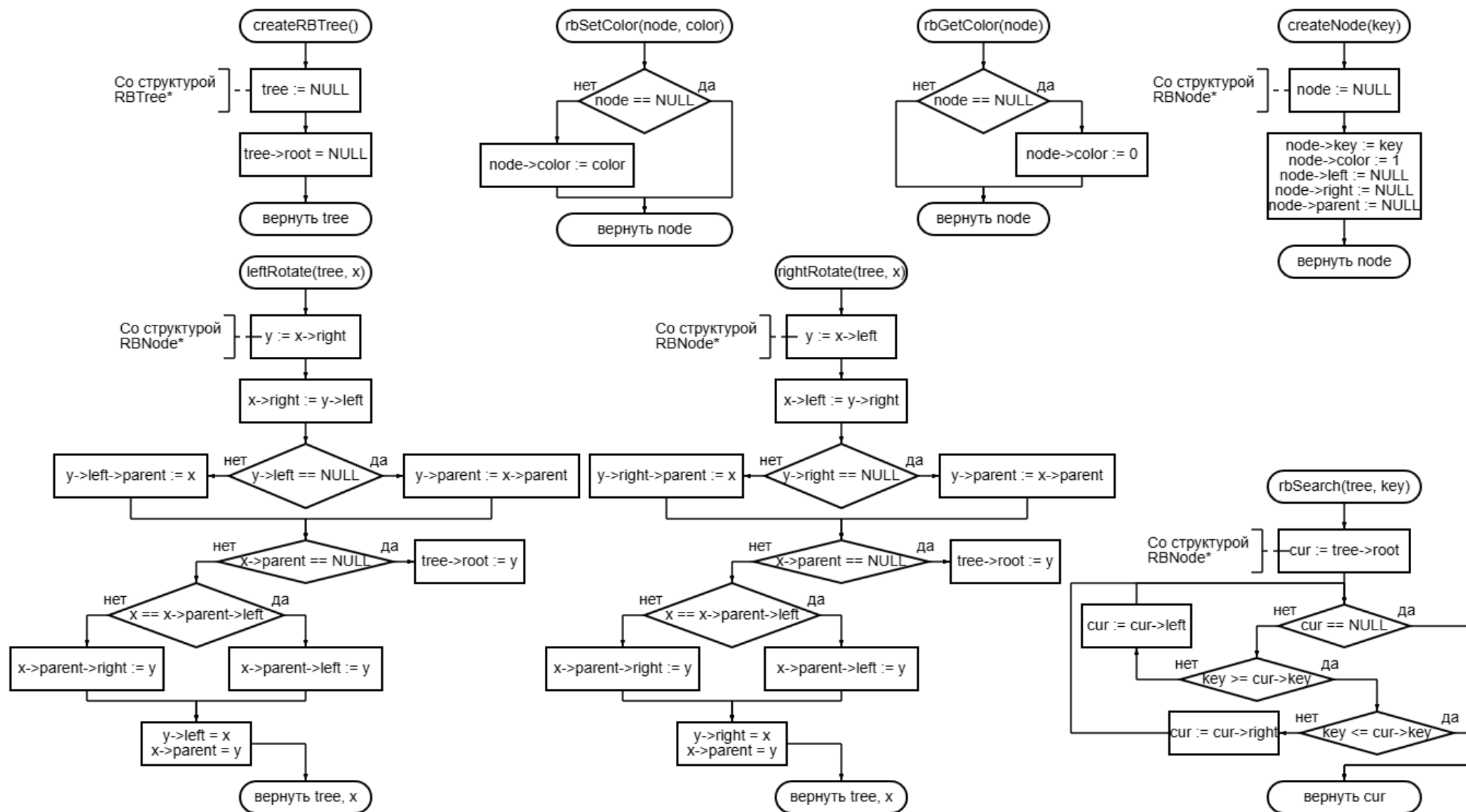




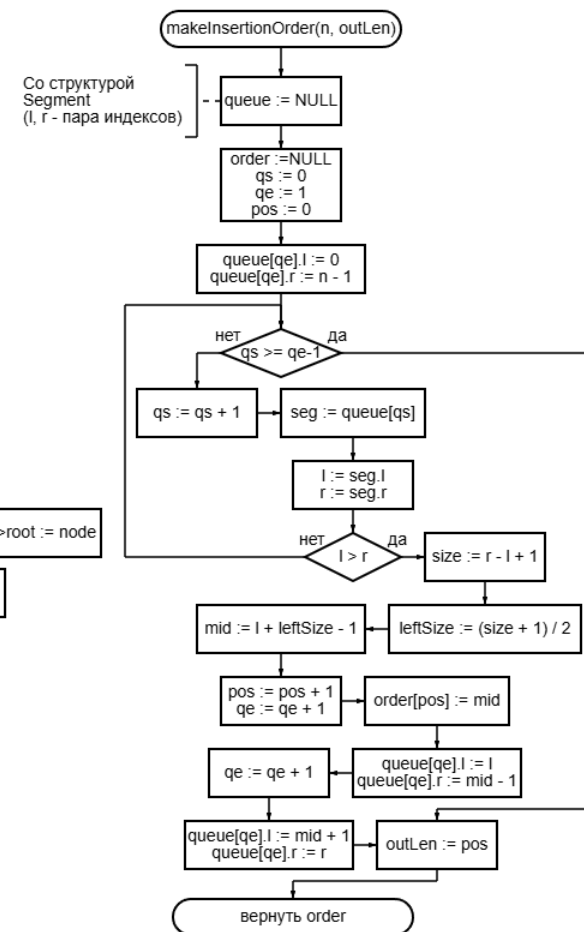
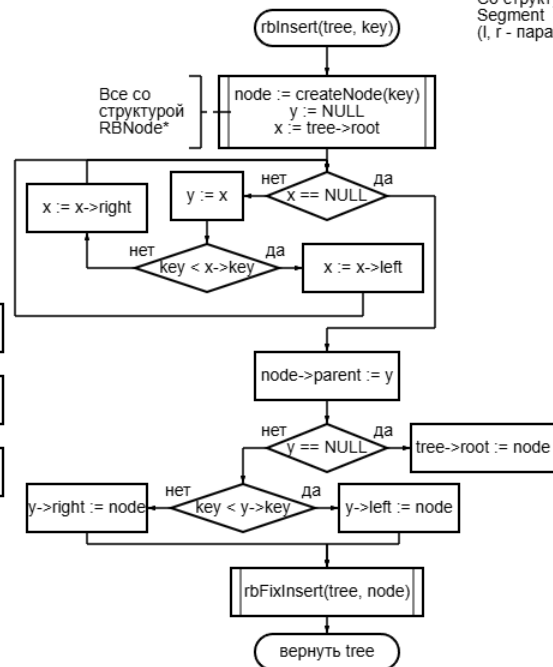
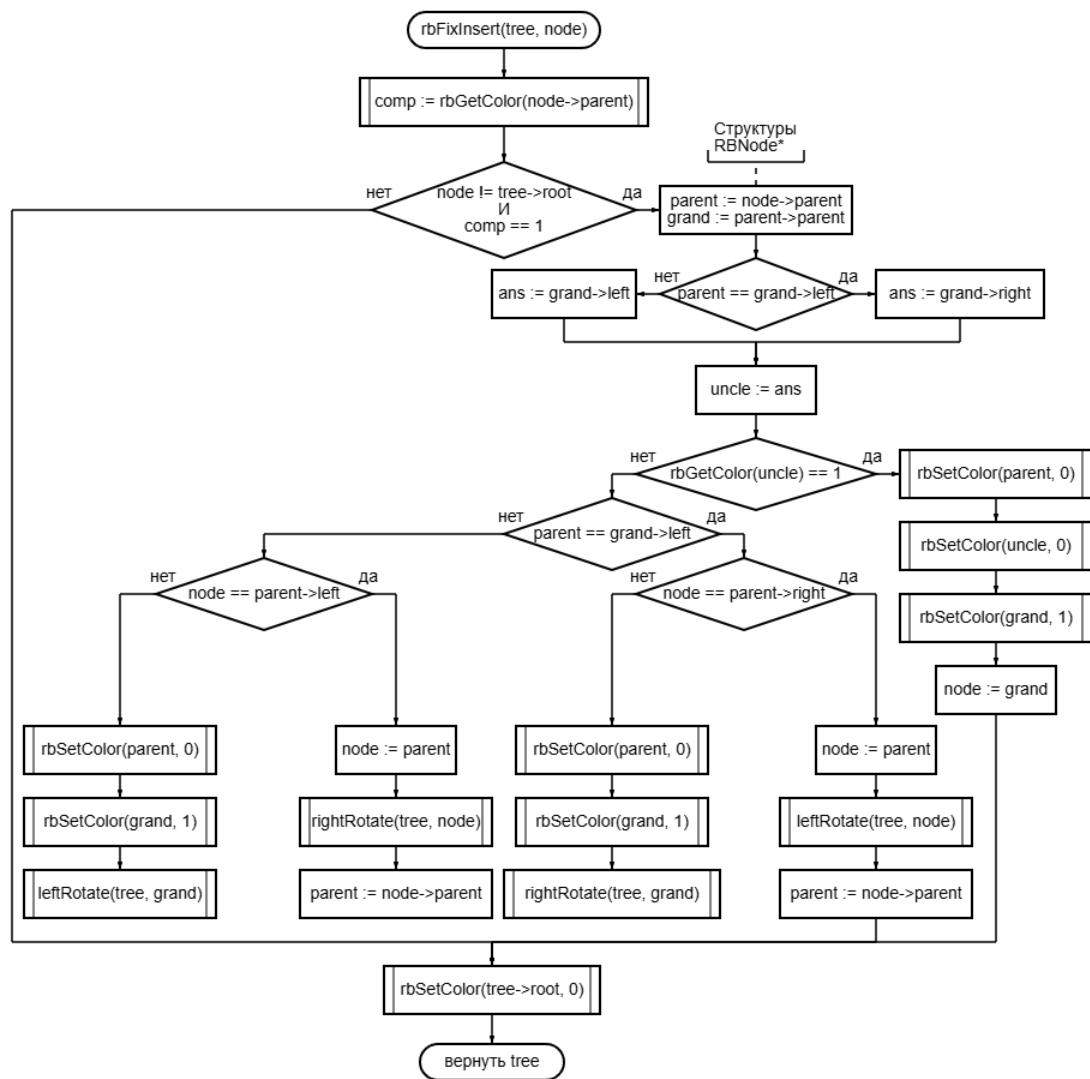
Подпрограммы (psrs)



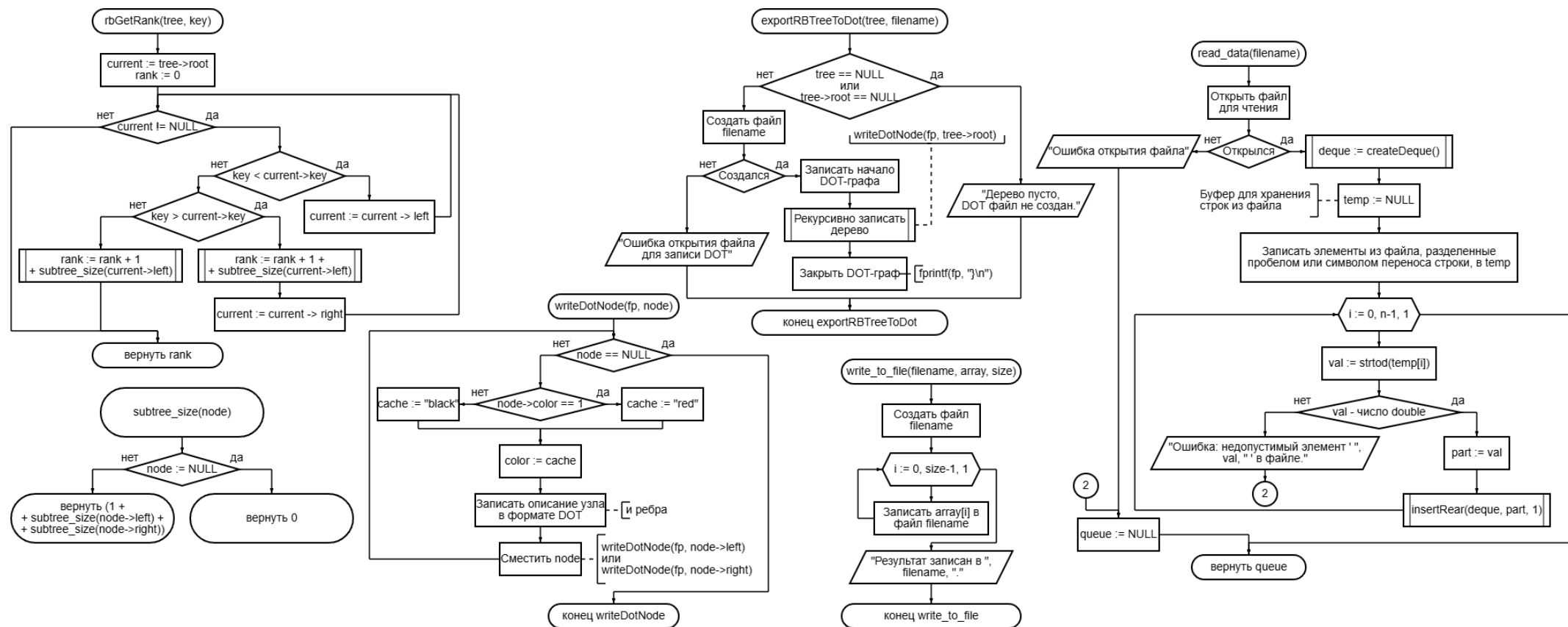
Подпрограммы (deque)



Подпрограммы (rbtree I)



Подпрограммы (rbtree II)



Подпрограммы (dot)

2 СПЕЦИФИКАЦИЯ ПЕРЕМЕННЫХ

Переменная	Описание	Тип использования	Тип	Размер (байт)	Диапазон значений
filename	Путь к входному файлу	Входная	const char*	8	Путь к файлу в файловой системе (строка)
p	Количество потоков	Входная	int	4	[2, 6]
splitters	Массив разделителей для многопутевого слияния	Промежуточная	double*	8	[1, 5]
sub_arrays	Массив указателей на подмассивы данных	Промежуточная	double**	8	Массив указателей на подмассивы
sub_sizes	Массив размеров подмассивов для каждого потока	Промежуточная	int*	4	[0, 2 147 483 647]
deque	Дек для хранения чисел	Промежуточная	Deque*	8	Указатель на структуру дека
n	Количество чисел в массиве	Промежуточная	int	4	[0, 2 147 483 647]
buffer	Массив для хранения элементов из очереди	Промежуточная	double*	8	[-1.7977e + 308, 1.7977e + 308]
index	Индекс текущего элемента в массиве	Промежуточная	int	4	[0, 2 147 483 646]
offset	Смещение для разбиения массива	Промежуточная	int	4	[0, 2 147 483 647]
size	Размер подмассива или блока	Промежуточная	int	4	[0, 2 147 483 647]
rem	Остаток элементов при делении на количество потоков	Промежуточная	int	4	[0, 5]
buckets	Массив указателей на дек для группировки данных	Промежуточная	Deque**	8	Указатели на структуры Deque
threads	Массив потоков	Промежуточная	pthread_t*	8	Системный указатель на поток (зависит от платформы)
tdata	Массив данных для потоков	Промежуточная	ThreadData*	8	Указатели на структуры ThreadData
samples	Массив выборок для вычисления разделителей	Промежуточная	double*	8	[2, 30]
command	Буфер для командной строки	Промежуточная	char [256]	256	Строка символов
tree	Красно-черное дерево	Промежуточная	RBTree*	8	Указатель на структуру дерева
insLen	Длина порядка вставки	Промежуточная	int	4	[0, 2 147 483 647]
insOrder	Порядок вставки в дерево	Промежуточная	int*	4	[0, 5]
dotFilename	Имя файла DOT	Выходная	cont char*	8	Путь к файлу (строка)
pngFilename	Имя файла PNG	Выходная	const char*	8	Путь к файлу (строка)

input	Ввод пользователя для поиска	Входная	char [64]	64	Строка символов
temp	Временный буфер для чтения из файла	Промежуточная	char [256]	256	Строка символов
val	Значение для поиска в дереве	Промежуточная	double	8	[-1.7977e + 308, 1.7977e + 308]
found	Найденный узел дерева	Промежуточная	RBNode*	8	Указатель на узел дерева
rank	Ранг найденного дерева	Промежуточная	int	4	[0, 2 147 483 647]
color	Цвет узла дерева	Промежуточная	const char*	8	«красный» или «черный»
RBNode.color	Цвет узла	Промежуточная	int	4	[0, 1]
array	Массив чисел для сортировки	Промежуточная	double*	8	[-1.7977e + 308, 1.7977e + 308]
id	Идентификаторы потока	Промежуточная	int	4	[0, 5]
prev	Указатель на предыдущий узел	Промежуточная	DequeNode*	8	Указатель или NULL
next	Указатель на следующий узел	Промежуточная	DequeNode*	8	Указатель или NULL
front	Указатель на первый узел дека	Промежуточная	DequeNode*	8	Указатель или NULL
rear	Указатель на последний узел дека	Промежуточная	DequeNode*	8	Указатель или NULL
key	Ключ узла дерева	Промежуточная	double	8	[-1.7977e + 308, 1.7977e + 308]
left	Указатель на левый дочерний узел	Промежуточная	RBNode*	8	Указатель или NULL
right	Указатель на правый дочерний узел	Промежуточная	RBNode*	8	Указатель или NULL
parent	Указатель на родительский узел	Промежуточная	RBNode*	8	Указатель или NULL
root	Указатель на корень дерева	Промежуточная	RBNode*	8	Указатель или NULL
l	Левая граница сегмента	Промежуточная	int	4	[0, 5]
r	Правая граница сегмента	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

Проект реализован на языке C и состоит из следующих файлов:

- `main.c` — основной файл с функцией `main()`;
- `psrs.c` — реализация многопоточной сортировки;
- `deque.c` — структура двусторонней очереди (дек);
- `rbtree.c` — реализация красно-чёрного дерева;
- `dot.c` — экспорт дерева в формате DOT (для визуализации с помощью Graphviz);
- `headers.h` — заголовочный файл с общими структурами и объявлениями функций;
- `Makefile` — файл сборки проекта.

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 <signal.h>

#define MAX_INPUT 64 // Максимальная длина вводимой строки
// Макросы для цветов в красно-черном дереве
#define RED 1
#define BLACK 0

// --- Структуры данных ---
// Структура для передачи данных в потоке
typedef struct {
    double* array;
    int size;
    int id;
} ThreadData;

// Узел дека
typedef struct DequeNode {
    double* array;
    int size;
    struct DequeNode* prev;
    struct DequeNode* next;
} DequeNode;

// Структура дека
typedef struct Deque {
    DequeNode* front;
    DequeNode* rear;
    int size;
} Deque;

// Узел красно-черного дерева
typedef struct RBNode {
    double key;
    int color;
    struct RBNode* left;
    struct RBNode* right;
    struct RBNode* parent;
} RBNode;

// Красно-черное дерево
typedef struct RBTree {
```

```

    RBNode* root;
} RBTree;

typedef struct {
    int l, r; // Пара индексов для сегмента
} Segment;

// --- Глобальные переменные ---
extern double* splitters; // Разделители
extern double** sub_arrays; // Подмассивы
extern int* sub_sizes; // Размеры подмассивов
extern volatile sig_atomic_t exit_flag;

// --- Функции работы с деком ---
Deque* createDeque();
void insertFront(Deque* dq, double* array, int size);
void insertRear(Deque* dq, double* array, int size);
DequeNode* deleteFront(Deque* dq);
DequeNode* deleteRear(Deque* dq);
int isEmptyDeque(Deque* dq);
void freeDeque(Deque* dq);

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

// --- Функции сортировки и распределения данных ---
int compare(const void* a, const void* b);
void* local_sort(void* arg);
void choose_samples(ThreadData* data, int p, double* samples);
void chooseSplitters(double* samples, int total_samples, int p);
void redistribute(int p);
void* merge_sort(void* arg);

void write_to_file(const char* filename, double* array, int size); // Запись в файл

// --- Функции работы с красно-черным деревом ---
int* makeInsertionOrder(int n, int* outLen); // Построение порядка вставки
RBTree* createRBTree(); // Создание красно-черного дерева
void rbInsert(RBTree* tree, double key); // Вставка в красно-черное дерево
RBNode* rbSearch(RBTree* tree, double key); // Поиск узла по ключу
void freeRBTree(RBTree* tree); // Освобождение памяти для дерева
int rbGetColor(RBNode* node); // Получение цвета узла
void rbSetColor(RBNode* node, int color); // Установка цвета узла
void leftRotate(RBTree* tree, RBNode* x); // Левый поворот
void rightRotate(RBTree* tree, RBNode* y); // Правый поворот
void rbFixInsert(RBTree* tree, RBNode* node); // Исправление дерева после вставки
int rbGetRank(RBTree* tree, double key); // Получение ранга узла по ключу
void exportRBTreeToDot(RBTree* tree, const char* filename); // Экспорт в DOT формат

// --- Функции для работы с файлами данных ---
Deque* read_data(const char* filename); // Чтение данных из файла

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

#endif // HEADERS_H

```

3.2.2 psrs.c

```

#include "headers.h"

// Comparator для qsort (возвращает -1,0,1)
int compare(const void *pa, const void *pb) {
    double a = *(const double*)pa;
    double b = *(const double*)pb;
    return (a > b) - (a < b);
}

void* local_sort(void *arg) {
    ThreadData *td = (ThreadData*)arg;
    qsort(td->array, td->size, sizeof(double), compare);
    printf("Поток %d получил %zu элементов. Отсортированный подмассив:\n",
        td->id + 1, (size_t)td->size);
    for (int i = 0; i < td->size; i++) {
        printf("%.15g ", td->array[i]);
    }
    printf("\n");
    return NULL;
}

```

```

void choose_samples(ThreadData *threads, int p, double *samples) {
    for (int i = 0; i < p; i++) {
        const ThreadData *td = &threads[i];
        // Берём p сэмплов равномерно из отсортированного фрагмента
        for (int j = 0; j < p; j++) {
            size_t idx = (size_t)j * td->size / p;
            samples[(size_t)i * p + j] = td->array[idx];
        }
    }
}

void choose_splitters(double *samples, int total_samples, int p) {
    puts("[Шаг 4] Сортировка вспомогательного массива выборок...");
    qsort(samples, total_samples, sizeof(double), compare);
    puts("[Шаг 5] Выбор разделителей для разбиения на группы...");
    // Каждый (k*p + p/2)-й элемент из отсортированных samples —
    // это медиана блока из p элементов
    for (int k = 1; k < p; k++) {
        splitters[k - 1] = samples[k * p + (p / 2) - 1];
    }
    printf("Выбранные разделители: ");
    for (int i = 0; i < p - 1; i++) {
        printf("%.15g ", splitters[i]);
    }
    printf("\n");
}

// Перераспределение по группам
void redistribute(int p) {
    // 1) Создать p деков — по одному для каждой группы
    Deque **buckets = calloc((size_t)p, sizeof(Deque*));
    if (!buckets) {
        perror("calloc(buckets)");
        exit(EXIT_FAILURE);
    }
    for (int i = 0; i < p; i++) {
        buckets[i] = createDeque();
        if (!buckets[i]) {
            fprintf(stderr, "Не удалось создать deque для группы %d\n", i);
            exit(EXIT_FAILURE);
        }
    }
    // 2) Пройти по каждому локально-отсортированному фрагменту
    for (int src = 0; src < p; src++) {
        double *arr = sub_arrays[src];
        int n = sub_sizes[src];

        for (int j = 0; j < n; j++) {
            double v = arr[j];
            int grp = 0;
            // найти первую splitters[grp] >= v
            while (grp < p - 1 && v > splitters[grp]) {
                grp++;
            }
            // Поместить v в дек buckets[grp]
            double *single = malloc(sizeof(double));
            if (!single) {
                perror("malloc(single)");
                exit(EXIT_FAILURE);
            }
            *single = v;
            insertRear(buckets[grp], single, 1);
        }
    }
    // 3) Собирать назад: каждый buckets[i] преобразовать в sub_arrays[i]
    for (int i = 0; i < p; i++) {
        // 3.1) посчитать итоговый размер
        size_t total = 0;
        for (DequeNode *node = buckets[i]->front; node; node = node->next) {
            total += node->size;
        }
        // 3.2) аллоцировать
        double *out = malloc(total * sizeof(double));
        if (!out) {
            perror("malloc(sub_arrays[i])");
            exit(EXIT_FAILURE);
        }
        // 3.3) заполнить и очистить deque
        size_t pos = 0;
    }
}

```

```

DequeNode *node;
while ((node = deleteFront(buckets[i])) != NULL) {
    memcpy(out + pos, node->array, node->size * sizeof(double));
    pos += node->size;
    free(node->array);
    free(node);
}
sub_arrays[i] = out;
sub_sizes[i] = (int)total;
freeDeque(buckets[i]);
}
free(buckets);
}

void* merge_sort(void *arg) {
    ThreadData *td = (ThreadData*)arg;
    qsort(td->array, td->size, sizeof(double), compare);
    return NULL;
}

```

3.2.3 deque.c

```

#include "headers.h"

Deque* createDeque() {
    Deque* dq = (Deque*)malloc(sizeof(Deque));
    dq->front = dq->rear = NULL;
    dq->size = 0;
    return dq;
}

int isEmptyDeque(Deque* dq) {
    return dq->front == NULL;
}

void insertFront(Deque* dq, double* array, int size) {
    DequeNode* newNode = (DequeNode*)malloc(sizeof(DequeNode));
    newNode->array = array;
    newNode->size = size;
    newNode->prev = NULL;
    newNode->next = dq->front;
    if (isEmptyDeque(dq)) {
        dq->front = dq->rear = newNode;
    } else {
        dq->front->prev = newNode;
        dq->front = newNode;
    }
    dq->size++;
}

void insertRear(Deque* dq, double* array, int size) {
    DequeNode* newNode = (DequeNode*)malloc(sizeof(DequeNode));
    newNode->array = array;
    newNode->size = size;
    newNode->next = NULL;
    if (isEmptyDeque(dq)) {
        dq->front = dq->rear = newNode;
        newNode->prev = NULL;
    } else {
        newNode->prev = dq->rear;
        dq->rear->next = newNode;
        dq->rear = newNode;
    }
    dq->size++;
}

DequeNode* deleteFront(Deque* dq) {
    if (isEmptyDeque(dq)) return NULL;
    DequeNode* temp = dq->front;
    dq->front = dq->front->next;
    if (dq->front) dq->front->prev = NULL;
    else dq->rear = NULL;
    dq->size--;
    return temp;
}

DequeNode* deleteRear(Deque* dq) {

```



```

    if (isEmptyDeque(dq)) return NULL;
    DequeNode* temp = dq->rear;
    dq->rear = dq->rear->prev;
    if (dq->rear) dq->rear->next = NULL;
    else dq->front = NULL;
    dq->size--;
    return temp;
}

void freeDeque(Deque* dq) {
    while (!isEmptyDeque(dq)) {
        DequeNode* node = deleteFront(dq);
        free(node->array);
        free(node);
    }
    free(dq);
}

// Чтение данных из файла в дек
Deque* read_data(const char* filename) {
    FILE* f = fopen(filename, "r");
    if (!f) {
        perror("Ошибка открытия файла");
        return NULL;
    }
    Deque* deque = createDeque();
    if (!deque) {
        fprintf(stderr, "Ошибка создания дека.\n");
        fclose(f);
        return NULL;
    }

    char temp[256];
    while (fscanf(f, "%255s", temp) == 1) {
        char* endptr;
        double val = strtod(temp, &endptr);
        if (endptr == temp || *endptr != '\0' || !isfinite(val)) {
            fprintf(stderr, "Ошибка: недопустимый элемент \"%s\" в файле.\n", temp);
            freeDeque(deque);
            fclose(f);
            return NULL;
        }
        double* part = malloc(sizeof(double));
        if (!part) {
            fprintf(stderr, "Ошибка выделения памяти.\n");
            freeDeque(deque);
            fclose(f);
            return NULL;
        }
        *part = val;
        insertRear(deque, part, 1);
    }
    fclose(f);
    return deque;
}

// Запись отсортированного массива в файл
void write_to_file(const char* filename, double* array, int size) {
    FILE* f = fopen(filename, "w");
    if (!f) {
        perror("Ошибка создания файла");
        return;
    }
    for (int i = 0; i < size; ++i) {
        fprintf(f, "%.15g ", array[i]);
    }
    printf("Результат записан в %s.\n\n", filename);
    fclose(f);
}

```

3.2.4 rbtree.c

```

#include "headers.h"

static RBNode* createNode(double key) {
    RBNode* node = (RBNode*)malloc(sizeof(RBNode));
    node->key = key;
}

```

```

    node->color = RED;
    node->left = NULL;
    node->right = NULL;
    node->parent = NULL;
    return node;
}

RBTree* createRBTree() {
    RBTree* tree = (RBTree*)malloc(sizeof(RBTree));
    tree->root = NULL;
    return tree;
}

int rbGetColor(RBNode* node) {
    return node ? node->color : BLACK;
}

void rbSetColor(RBNode* node, int color) {
    if (node) node->color = color;
}

void leftRotate(RBTree* tree, RBNode* x) {
    RBNode* y = x->right;
    x->right = y->left;
    if (y->left) y->left->parent = x;
    y->parent = x->parent;
    if (!x->parent)
        tree->root = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;
    y->left = x;
    x->parent = y;
}

void rightRotate(RBTree* tree, RBNode* y) {
    RBNode* x = y->left;
    y->left = x->right;
    if (x->right) x->right->parent = y;
    x->parent = y->parent;
    if (!y->parent)
        tree->root = x;
    else if (y == y->parent->left)
        y->parent->left = x;
    else
        y->parent->right = x;
    x->right = y;
    y->parent = x;
}

void rbFixInsert(RBTree* tree, RBNode* node) {
    while (node != tree->root && rbGetColor(node->parent) == RED) {
        RBNode* parent = node->parent;
        RBNode* grand = parent->parent;
        if (!grand) break; // защита
        RBNode* uncle = (parent == grand->left) ? grand->right : grand->left;

        if (rbGetColor(uncle) == RED) {
            rbSetColor(parent, BLACK);
            rbSetColor(uncle, BLACK);
            rbSetColor(grand, RED);
            node = grand;
        } else {
            if (parent == grand->left) {
                if (node == parent->right) {
                    node = parent;
                    leftRotate(tree, node);
                    parent = node->parent;
                }
                rbSetColor(parent, BLACK);
                rbSetColor(grand, RED);
                rightRotate(tree, grand);
            } else {
                if (node == parent->left) {
                    node = parent;
                    rightRotate(tree, node);
                    parent = node->parent;
                }
            }
        }
    }
}

```

```

        }
        rbSetColor(parent, BLACK);
        rbSetColor(grand, RED);
        leftRotate(tree, grand);
    }
    rbSetColor(tree->root, BLACK);
}

void rbInsert(RBTree* tree, double key) {
    RBNode* node = createNode(key);
    RBNode* y = NULL;
    RBNode* x = tree->root;
    while (x) {
        y = x;
        if (key < x->key)
            x = x->left;
        else
            x = x->right;
    }
    node->parent = y;
    if (!y)
        tree->root = node;
    else if (key < y->key)
        y->left = node;
    else
        y->right = node;
    rbFixInsert(tree, node);
}

RBNode* rbSearch(RBTree* tree, double key) {
    RBNode* cur = tree->root;
    while (cur) {
        if (key < cur->key)
            cur = cur->left;
        else if (key > cur->key)
            cur = cur->right;
        else
            return cur;
    }
    return NULL;
}

static void freeRBNode(RBNode* node) {
    if (!node) return;
    freeRBNode(node->left);
    freeRBNode(node->right);
    free(node);
}

void freeRBTree(RBTree* tree) {
    if (!tree) return;
    freeRBNode(tree->root);
    free(tree);
}

int* makeInsertionOrder(int n, int* outLen) {
    Segment *queue = malloc(sizeof(Segment) * (2 * n + 1));
    int *order = malloc(sizeof(int) * n);
    int qs = 0, qe = 0, pos = 0;

    queue[qe++] = (Segment){ .l = 0, .r = n - 1 };
    while (qs < qe) {
        Segment seg = queue[qs++];
        int l = seg.l, r = seg.r;
        if (l > r) continue;

        // длина отрезка
        int size = r - l + 1;
        // размер "левой" половины: при нечётном size
        int leftSize = (size + 1) / 2;
        // индекс медианы (первого элемента правой части)
        int mid = l + leftSize - 1;

        order[pos++] = mid;

        queue[qe++] = (Segment){ .l = l, .r = mid - 1 };
        queue[qe++] = (Segment){ .l = mid + 1, .r = r };
    }
}

```

```

    free(queue);

    *outLen = pos;
    return order;
}

```

3.2.5 dot.c

```

#include "headers.h"

// Вспомогательная функция для вычисления размера поддерева с корнем в узле
static int subtree_size(RBNode* node) {
    if (node == NULL) return 0;
    return 1 + subtree_size(node->left) + subtree_size(node->right);
}

// Вычисление ранга (1-основанный) узла с данным ключом в дереве
int rbGetRank(RBTree* tree, double key) {
    RBNode* current = tree->root;
    int rank = 0;

    // Поиск узла с заданным ключом
    while (current != NULL) {
        if (key < current->key) {
            current = current->left;
        } else if (key > current->key) {
            rank += 1 + subtree_size(current->left);
            current = current->right;
        } else {
            rank += subtree_size(current->left) + 1;
            return rank; // Возвращаем ранг узла
        }
    }
    return -1; // Ключ не найден
}

// Вспомогательная функция для записи узла и рёбер в DOT файл
static void writeDotNode(FILE* fp, RBNode* node) {
    if (node == NULL) {
        return;
    }

    const char* color = (node->color == RED) ? "red" : "black";

    // Записываем описание узла в формате DOT
    fprintf(fp,
        "    \n%p\n" [label=\ "%15g\n", color=%s, fontcolor=%s, style=filled, fillcolor=white,
        shape=circle];\n",
        (void*)node, node->key, color, color);

    if (node->left) {
        fprintf(fp, "    \n%p\n" -> \ "%p\n";\n", (void*)node, (void*)node->left);
        writeDotNode(fp, node->left);
    } else {
        fprintf(fp,
            "    null%pL [label=\ "NIL\n", shape=box, style=filled, fillcolor=gray];\n",
            (void*)node);
        fprintf(fp, "    \n%p\n" -> null%pL [style=dotted];\n",
            (void*)node, (void*)node);
    }

    if (node->right) {
        fprintf(fp, "    \n%p\n" -> \ "%p\n";\n", (void*)node, (void*)node->right);
        writeDotNode(fp, node->right);
    } else {
        fprintf(fp,
            "    null%pR [label=\ "NIL\n", shape=box, style=filled, fillcolor=gray];\n",
            (void*)node);
        fprintf(fp, "    \n%p\n" -> null%pR [style=dotted];\n",
            (void*)node, (void*)node);
    }
}

void exportRBTreeToDot(RBTree* tree, const char* filename) {
    if (tree == NULL || tree->root == NULL) {
        fprintf(stderr, "Дерево пусто, DOT файл не создан.\n");
        return;
    }
}

```

```

FILE* fp = fopen(filename, "w");
if (!fp) {
    perror("Ошибка открытия файла для записи DOT");
    return;
}

// Записываем начало DOT-графа
fprintf(fp, "digraph RBTREE {\n");
fprintf(fp, "    node [fontname=\"Arial\"]; \n");

// Рекурсивная запись дерева
writeDotNode(fp, tree->root);

// Закрытие DOT-графа
fprintf(fp, "}\n");
fclose(fp);
}

```

3.2.6 main.c

```

#include "headers.h"

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

void handle_sigint(int sig) {
    (void) sig;
    exit_flag = 1;
}

int main(int argc, char* argv[]) {
    signal(SIGINT, handle_sigint);
    // Проверка аргументов командной строки
    if (argc != 3) {
        printf("Использование: %s <входной_файл> <количество_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;
    }
    // Запись данных из файла в дек
    Deque* deque = read_data(filename);
    if (!deque) {
        return 1;
    }
    int n = deque->size;
    printf("Прочитано %d чисел из файла.\nСодержимое дека:\n", n);
    // Печать содержимого дека
    for (DequeNode* node = deque->front; node; node = node->next) {
        printf("%.15g ", *node->array);
    }
    printf("\n\n");
    if (n == 0) {
        fprintf(stderr, "Ошибка: файл не содержит double числа.\n");
        freeDeque(deque);
        return 1;
    }
    double* buffer = malloc(n * sizeof(double));
    if (!buffer) {
        fprintf(stderr, "Ошибка выделения памяти для массива.\n");
        freeDeque(deque);
        return 1;
    }
    // Заполнение массива из дека
    int index = 0;
    while (!isEmptyDeque(deque)) {
        DequeNode* dnode = deleteFront(deque);
        buffer[index++] = *dnode->array;
        free(dnode->array);
        free(dnode);
    }
}

```

```

freeDeque(deque);

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);
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 };
    pthread_create(&threads[i], NULL, local_sort, &tdata[i]);
    offset += size;
}
for (int i = 0; i < p; ++i) pthread_join(threads[i], NULL);
printf("[Шаг 3] Создание вспомогательного массива выборок...\n");
double* samples = malloc(p * p * sizeof(double));
choose_samples(tdata, p, samples);
splitters = malloc((p - 1) * sizeof(double));
chooseSplitters(samples, p * p, p);
free(samples);
// Разбиение данных по разделителям
printf("[Шаг 6] Разбиение данных в каждом потоке согласно разделителям...\n");
redistribute(p);
// Многопутевое слияние
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]);
}
for (int i = 0; i < p; ++i) pthread_join(threads[i], NULL);
// Объединение отсортированных частей
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];
}
// Вывод отсортированного массива
printf("\nОтсортированный массив:\n");
for (int i = 0; i < n; ++i) printf("%.15g ", result[i]);
printf("\n\n");
write_to_file("sorted_result.txt", result, n);
char command[256];
// Открытие результата
snprintf(command, sizeof(command), "sub1 sorted_result.txt");
if (system(command) != 0) {
    printf("Ошибка при sorted_result.txt.\n");
}
// Работа с красно-черным деревом
RBTREE* tree = createRBTREE();
int insLen;
int *insOrder = makeInsertionOrder(n, &insLen);
for (int i = 0; i < insLen; ++i) {
    int idx = insOrder[i];
    rbInsert(tree, result[idx]);
}
free(insOrder);
// Освобождение памяти
for (int i = 0; i < p; ++i) free(sub_arrays[i]);
free(sub_arrays);
free(sub_sizes);
free(splitters);
free(threads);
free(tdata);
free(result);

```

```

free(buffer);
// Экспорт дерева в DOT файл и создание PNG
const char* dotFilename = "tree.dot";
const char* pngFilename = "tree.png";
exportRBTreeToDot(tree, dotFilename);
printf("DOT файл дерева сохранён в %s\n", dotFilename);
// Генерация PNG с помощью Graphviz
snprintf(command, sizeof(command), "dot -Tpng %s -o %s", dotFilename, pngFilename);
int ret = system(command);
if (ret == 0) {
    printf("PNG изображение дерева успешно создано: %s\n", pngFilename);
    // Открытие PNG файла
    snprintf(command, sizeof(command), "xdg-open %s", pngFilename);
    if (system(command) != 0) {
        printf("Ошибка при открытии PNG изображения.\n");
    }
} else {
    printf("Ошибка: Убедитесь, что Graphviz установлен.\nДоступна ли команда dot -Tpng...?\n");
}
// Цикл для поиска числа в красно-черном дереве
char input[MAX_INPUT];
while (1) {
    if (exit_flag) {
        printf("Программа остановлена.\n");
        break;
    }
    printf("\nВведите число для поиска в красно-черном дереве ('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;
    }
    char* endptr;
    double val = strtod(input, &endptr);
    if (endptr == input || (*endptr != '\n' && *endptr != '\0')) {
        printf("Некорректный ввод. Введите число.\n");
        continue;
    }
    RBNode* found = rbSearch(tree, val);
    if (found) {
        int rank = rbGetRank(tree, val);
        const char* color = (found->color == 1) ? "красный" : "черный";
        printf("Позиция (ранг) числа %.15g в дереве: %d, цвет узла: %s\n",
            val, rank, color);
    } else {
        printf("Число %.15g не найдено в дереве.\n", val);
    }
}
freeRBTree(tree);
return 0;
}

```

3.2.7 Makefile

```

CC = gcc
CFLAGS = -Wall -Wextra -Werror -pthread -O3
LDFLAGS = -lm

SRC = main.c deque.c psrs.c rbtree.c dot.c
OBJ = $(SRC:.c=.o)
TARGET = psrs

all: $(TARGET)

$(TARGET): $(OBJ)
    $(CC) $(CFLAGS) -o $(TARGET) $(OBJ) $(LDFLAGS)

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

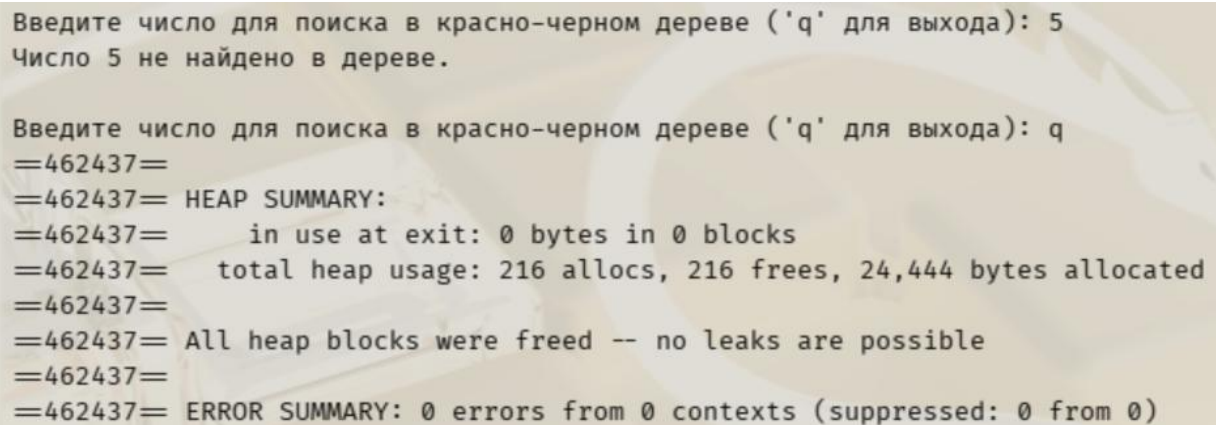
clean:
    rm -f $(OBJ) $(TARGET)

```

4 ТЕСТИРОВАНИЕ ПРОГРАММЫ

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

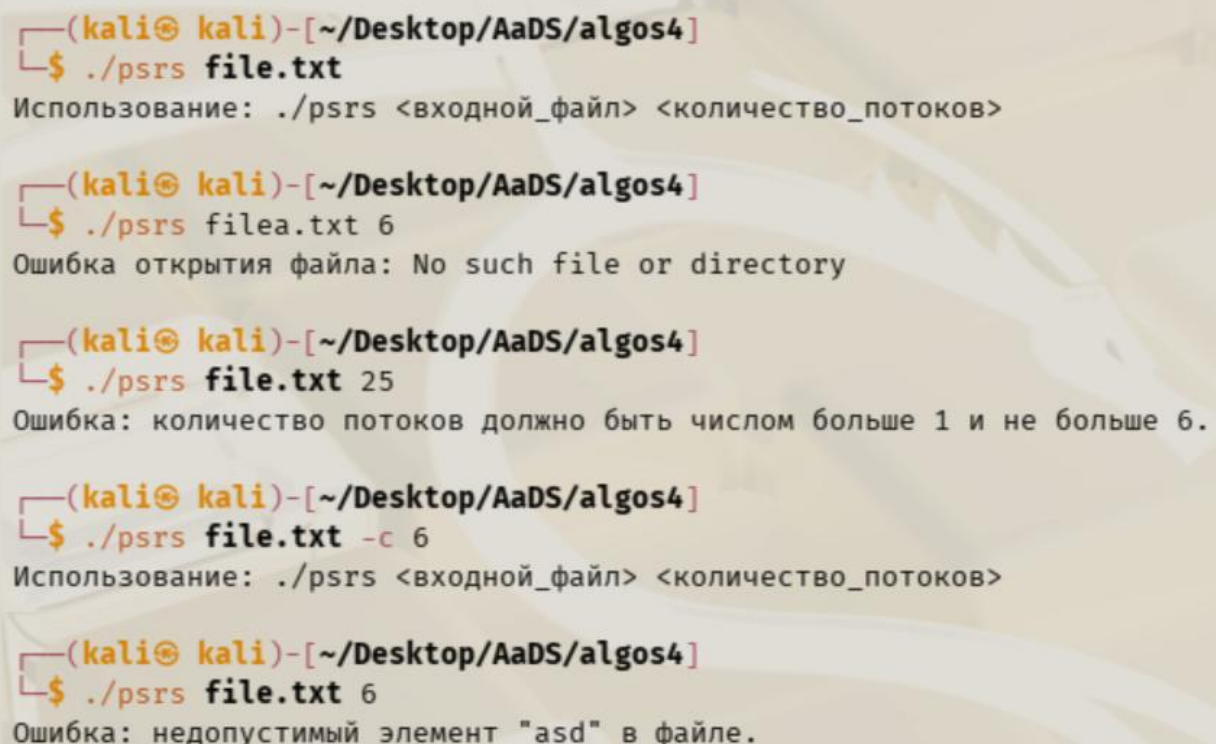
4.1 Скриншоты тестирования



```
Введите число для поиска в красно-черном дереве ('q' для выхода): 5
Число 5 не найдено в дереве.

Введите число для поиска в красно-черном дереве ('q' для выхода): q
=462437=
=462437= HEAP SUMMARY:
=462437=      in use at exit: 0 bytes in 0 blocks
=462437=    total heap usage: 216 allocs, 216 frees, 24,444 bytes allocated
=462437=
=462437= All heap blocks were freed -- no leaks are possible
=462437=
=462437= ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Рисунок 1 – Valgrind



```
(kali@ kali)-[~/Desktop/AaDS/algos4]
$ ./psrs file.txt
Использование: ./psrs <входной_файл> <количество_потоков>

(kali@ kali)-[~/Desktop/AaDS/algos4]
$ ./psrs filea.txt 6
Ошибка открытия файла: No such file or directory

(kali@ kali)-[~/Desktop/AaDS/algos4]
$ ./psrs file.txt 25
Ошибка: количество потоков должно быть числом больше 1 и не больше 6.

(kali@ kali)-[~/Desktop/AaDS/algos4]
$ ./psrs file.txt -c 6
Использование: ./psrs <входной_файл> <количество_потоков>

(kali@ kali)-[~/Desktop/AaDS/algos4]
$ ./psrs file.txt 6
Ошибка: недопустимый элемент "asd" в файле.
```

Рисунок 2 – Обработка ошибок


```

L$ ./psrs file.txt 6
Прочитано 35 чисел из файла.
Содержимое дека:
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 9999 1000000000
0 1.23e+45 3.14159e-05 1.7976e+308 -1.7976e+308 1e-308

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 1 получил 6 элементов. Отсортированный подмассив:
Поток 5 получил 6 элементов. Отсортированный подмассив:
0 -0.5 0 100 1000 9999 10000000000
Поток 4 получил 6 элементов. Отсортированный подмассив:
0 100000 1000000 10000000 100000000 1000000000 1000000000
1 Поток 3 получил 6 элементов. Отсортированный подмассив:
1.2 5.3 450.5454654 1e+300
Поток 2 получил 6 элементов. Отсортированный подмассив:
1e-300 1e-100 1 42 1e+100 1e+300
Поток 6 получил 5 элементов. Отсортированный подмассив:
-1.7976e+308 1e-308 3.14159e-05 1.23e+45 1.7976e+308
-1 1.23 2 3 1000 10000
[Шаг 3] Создание вспомогательного массива выборов ...
[Шаг 4] Сортировка вспомогательного массива выборов ...
[Шаг 5] Выбор разделителей для разбиения на группы ...
Выбранные разделители: 1e-300 1.23 450.5454654 1000000 1e+100
[Шаг 6] Разбиение данных в каждом потоке согласно разделителям ...
[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 100000000 1000000000 10000000000
10000000000 1.23e+45 1e+100 1e+300 1e+300 1.7976e+308

Результат записан в sorted_result.txt.

DOT файл дерева сохранён в tree.dot
PNG изображение дерева успешно создано: tree.png

Введите число для поиска в красно-черном дереве ('q' для выхода): -1
Позиция (ранг) числа -1 в дереве: 2, цвет узла: черный

Введите число для поиска в красно-черном дереве ('q' для выхода): █

```

Рисунок 3 – Пример работы программы

```

Введите число для поиска в красно-черном дереве ('q' для выхода): -1
Позиция (ранг) числа -1 в дереве: 2, цвет узла: черный

Введите число для поиска в красно-черном дереве ('q' для выхода): 1
Позиция (ранг) числа 1 в дереве: 11, цвет узла: черный

Введите число для поиска в красно-черном дереве ('q' для выхода): 1.23
Позиция (ранг) числа 1.23 в дереве: 14, цвет узла: черный

Введите число для поиска в красно-черном дереве ('q' для выхода): asd
Некорректный ввод. Введите число.

Введите число для поиска в красно-черном дереве ('q' для выхода): -5
Число -5 не найдено в дереве.

Введите число для поиска в красно-черном дереве ('q' для выхода): 42
Позиция (ранг) числа 42 в дереве: 18, цвет узла: черный

Введите число для поиска в красно-черном дереве ('q' для выхода): 1e+300
Позиция (ранг) числа 1e+300 в дереве: 33, цвет узла: красный

Введите число для поиска в красно-черном дереве ('q' для выхода): 1e+301
Число 1e+301 не найдено в дереве.

Введите число для поиска в красно-черном дереве ('q' для выхода): q

```

Рисунок 4 – Поиск в красно-черном дереве

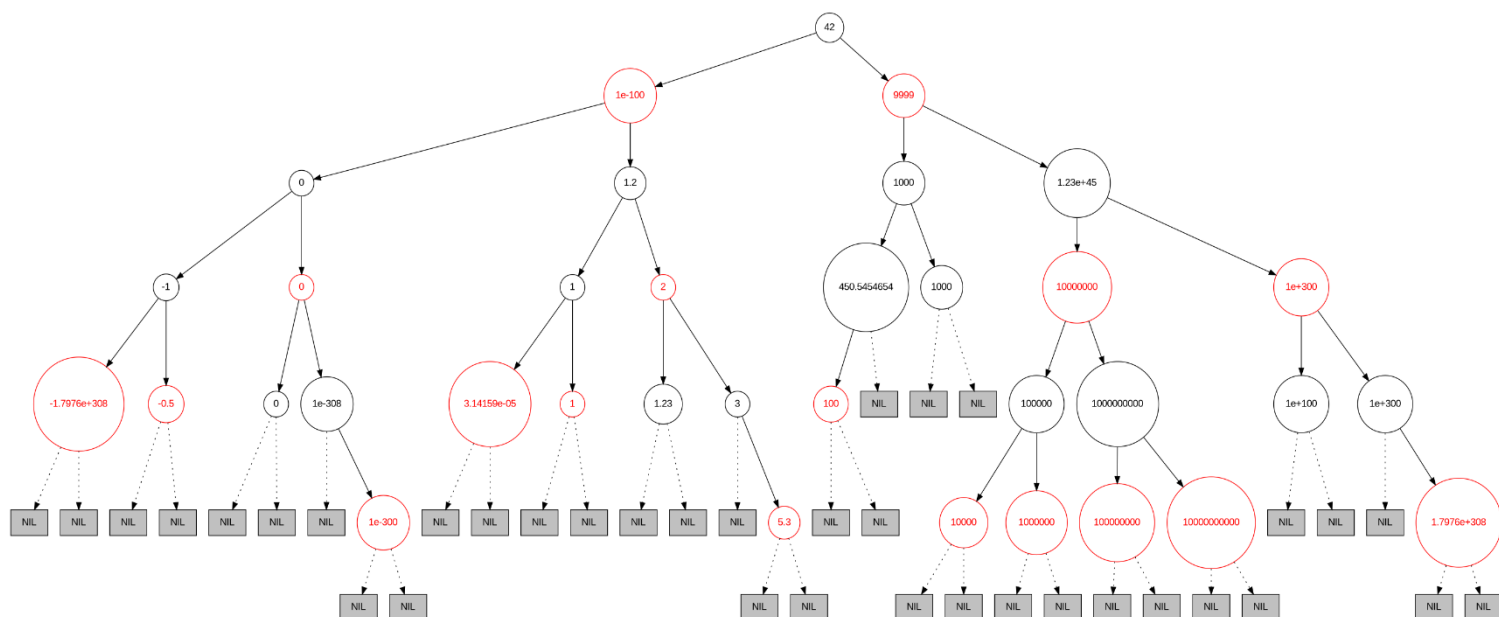


Рисунок 5 – Пример симметричного красно-черного дерева

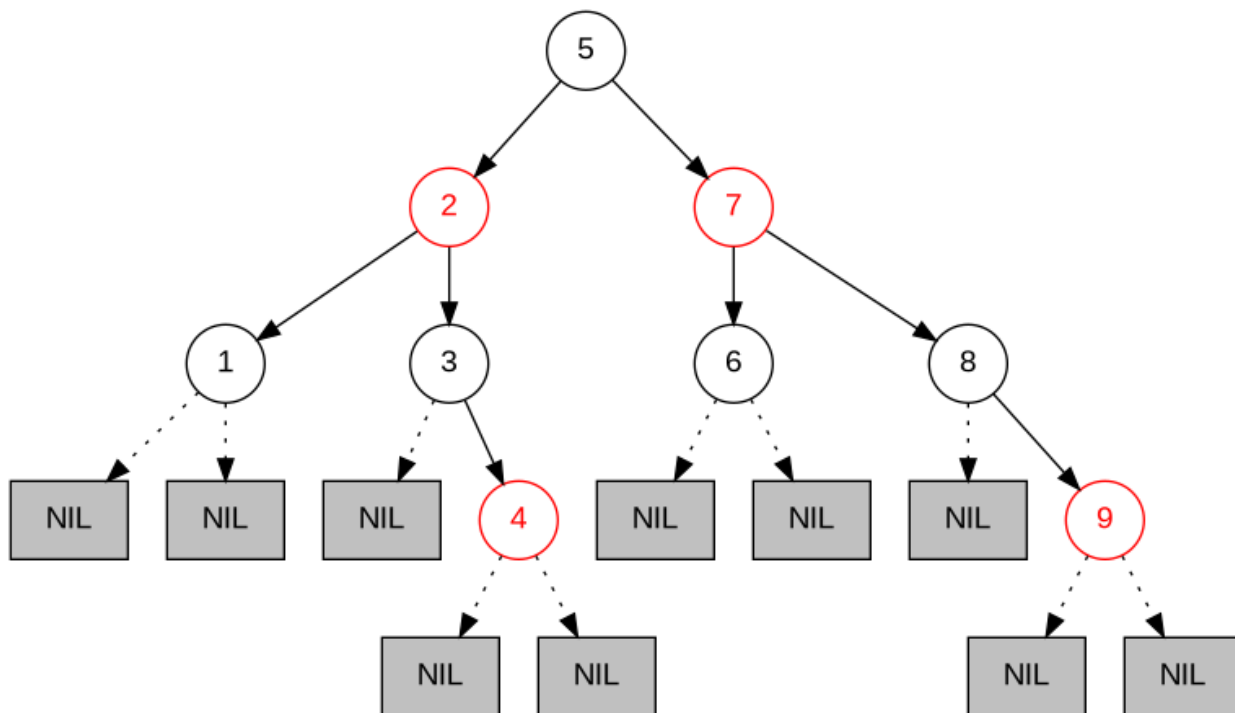


Рисунок 6 – Простой пример красно-черного дерева

ЗАКЛЮЧЕНИЕ

В ходе лабораторной работы была разработана и реализована программа многопоточной сортировки методом PSRS с использованием дека на базе связного списка и последующей записью результата в красно-чёрное дерево.

Алгоритм PSRS позволил эффективно распределить работу между потоками, обеспечив ускорение сортировки. Теоретическая сложность алгоритма составляет $O\left(\frac{n}{p} \log \frac{n}{p}\right)$ при использовании p потоков, что делает его масштабируемым решением для сортировки больших массивов.

Для хранения отсортированных значений применено красно-чёрное дерево, обладающее следующими свойствами: логарифмическая сложность операций вставки и поиска, устойчивость к вырожденным случаям и гарантия сбалансированности. Сложность вставки и поиска в красно-чёрном дереве составляет $O(\log n)$.

Выполненная работа позволила закрепить навыки многопоточного программирования, работы со структурами данных и динамической памятью.