

**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 №2

«Пирамидальная сортировка и поиск методом КМП»

Выполнил:

Заговенко Всеволод Романович, студент группы N3249

(подпись)

Проверил:

Ерофеев Сергей Анатольевич, преподаватель

(отметка о выполнении)

Санкт-Петербург

2025 г.

Постановка Задачи

Задача: Разработать программу пирамидальной сортировки с последующим поиском методом Кнута-Морриса-Пратта, используя дек на базе связного списка. Оценить сложность по Фофанову.

Техническое задание

1. Описание алгоритма

В программе используется класс Node, который отвечает за элементы дека. Реализацией самого дека и его методов занимается класс Deque.

Методы класса Deque:

- push_front – добавление элемента в начало дека.
- push_back – добавление элемента в конец дека.
- pop_front – удаление элемента из начала дека.
- pop_back – удаление элемента из конца дека.

Для упрощенной работы с деком реализованы вспомогательные функции:

- size_deq – вычисляет размер дека.
- getLn – возвращает узел по индексу.
- insert – вставляет элемент в дека на указанную позицию.

Помимо выше перечисленного, в программе реализованы функции для пирамидальной сортировки:

- max_heap – преобразует поддерево в максимальную кучу (наибольший элемент в корне).
- pyramid_sort – сортирует дека с помощью пирамидальной сортировки.

Функция для метода КМП:

- KMP_SEQ

Описание алгоритма самой работы программы:

Основная программа(Main) начинается с выбора способа ввода данных в дек: ввод осуществляется либо из файла, либо из стандартного ввода. После этого мы вводим образ, который хотим найти в исходном деке и в отсортированном. Затем происходит проверка на пустоту деков. После этого мы вызываем функцию KMP_SEQ для поиска образа в исходном деке.

I Реализация KMP_SEQ:

1. Инициализация префикс-функции:

1.1. Создается дополнительный дек (prefix) для хранения значений префикс-функции.

1.2. Заполняется нулями (длиной равной образцу img).

2. Вычисление префикс-функции:

2.1. Сравнивается образец сам с собой, начиная со второго элемента ($i=1, j=0$)

2.2. При совпадении элементов ($ptr1 \rightarrow data == ptr2 \rightarrow data$):

2.2.1. Записываем $j+1$ в $prefix[i]$ (длина совпавшего префикса).

2.2.1. Увеличиваем оба индекса ($i++, j++$).

2.3 При несовпадении:

2.2.1. Если $j=0$ - записываем 0 в $prefix[i]$ и увеличиваем i .

2.2.1. Иначе - сдвигаем j назад по уже вычисленной префикс-функции ($j = prefix[j-1]$).

3. Поиск образа в последовательности:

3.1. Устанавливаем начальные индексы ($i=0$ для последовательности, $j=0$ для образа).

3.2. Поэлементное сравнение:

3.2.1. При совпадении - двигаемся дальше ($i++, j++$).

3.2.2. При полном совпадении ($j == m$) - выводим позицию начала совпадения ($i-j$) и выходим из функции.

3.2.3. При несовпадении:

3.2.3.1. Если $j>0$ - сдвигаем j по префикс-функции ($j = prefix[j-1]$).

3.2.3.1. Иначе - просто увеличиваем i .

4. Завершение:

4.1. Если дошли до конца последовательности ($i == n$) - выводим "image not find".

Как только функция KMP_SEQ завершила свою работу, мы вызываем функцию `pyramid_sort`, чтобы отсортировать наш дек.

| Реализация `pyramid_sort`:

1. Подготовка:

1.1. Создается временный узел (`ptr_tmp`) для обмена значений.

1.2. Запоминается исходный размер дека (`n_size`).

2. Основной цикл сортировки:

2.1. Выполняется, пока в неотсортированной части больше 1 элемента.

2.2. Каждая итерация включает:

2.2.1. Построение максимальной кучи с помощью функции `max_heap`.

2.2.2. Обмен первого элемента (максимум) с последним в неотсортированной части.

2.2.3. Уменьшение размера неотсортированной части (`n_size=n_size-1`).

3. Завершение

3.1 Освобождается память временного узла.

| Реализация `max_heap`:

1.Инициализация:

1.1. Создаются временные переменные для хранения значений и указателей на узлы.

1.2. Устанавливается флаг `check_heap` в `true` для первого прохода.

2. Основной цикл:

2.1. Выполняется, пока куча не будет полностью упорядочена (`check_heap = false`).

2.2 Проход начинается с последнего родительского узла (индекс $n_size/2-1$) до корня (индекс 0).

3. Обработка каждого узла:

3.1 Для текущего родительского узла (ptr1):

3.1.1. Запоминается его значение (value_leaf).

3.1.2. Проверяется существование левого потомка (индекс $i*2+1$).

3.1.3. Если левый потомок существует, сравнивается его значение с родителем.

3.1.4. Проверяется существование правого потомка (индекс $i*2+2$).

3.1.5. Если правый потомок существует, сравнивается его значение с текущим максимумом.

4.Обмен значений:

4.1Если найден потомок с большим значением:

4.1.1 Происходит обмен значений между родителем и этим потомком.

4.1.2. Устанавливается флаг `check_heap = true` (требуется повторная проверка кучи).

После того как мы отсортировали наш дек, мы выводим его и затем вызываем функцию KMP_SEQ для поиска образца уже в отсортированном deque. (Реализация KMP_SEQ в самом начале описания алгоритма программы.)

2.Переменные, используемые в программе.

Примечание:

Емкость указателей считается в отношении x86/x64.

2.1 Глобальные переменные:

Примечание:

Память под переменные `deq` и `img_deq` выделяется динамически, и размер диапазона зависит от количества введенных данных. Поэтому диапазон обозначим как `N`.

Название переменной	Описание	Тип	Диапазон	Ёмкость
<code>deq</code>	Основной дек.	<code>Deque</code>	<code>N</code>	<code>N*12</code> байт
<code>img_deq</code>	Дек для образа.	<code>Deque</code>	<code>N</code>	<code>N*12</code> байт

2.2 Класс `Node`:

Название переменной	Описание	Тип	Диапазон(x86)	Ёмкость(байт)
<code>data</code>	Значение хранимое в узле.	<code>float</code>	- 3.4E+38 до +3.4E+38	4
<code>next</code>	Указатель на следующий узел.	<code>Node*</code>	0x00000000 до 0xFFFFFFFF	4/8
<code>prev</code>	Указатель на предыдущий узел.	<code>Node*</code>	0x00000000 до 0xFFFFFFFF	4/8

2.3 Класс Deque:

Название переменной	Описание	Тип	Диапазон(х86)	Ёмкость(байт)
head	Указатель на начало дека.	Node*	0x00000000 до 0xFFFFFFFF	4/8
tail	Указатель на конец дека.	Node*	0x00000000 до 0xFFFFFFFF	4/8

2.4 Функция size_deq:

Название переменной	Описание	Тип	Диапазон(х86)	Ёмкость(байт)
k	Счетчик элементов.	int	0 до $2^{31} - 1$	4
ptr	Текущий узел при обходе.	Node*	0x00000000 до 0xFFFFFFFF	4/8

2.5 Функция getLn:

Название переменной	Описание	Тип	Диапазон(х86)	Ёмкость(байт)
n	Счетчик при поиске.	int	0 до $2^{31} - 1$	4
ptr	Текущий узел при поиске.	Node*	0x00000000 до 0xFFFFFFFF	4/8

2.6 Функция print_deck:

Название переменной	Описание	Тип	Диапазон(х86)	Ёмкость(байт)
ptr	Текущий узел при обходе.	Node*	0x00000000 до 0xFFFFFFFF	4/8

2.7 Функция main

Название переменной	Описание	Тип	Диапазон(х86)	Ёмкость(байт)
x	Входное значение данных.	float	- 3.4E+38 до +3.4E+38	4
mode	Режим ввода данных.	int	-2 ³¹ до 2 ³¹ -1	4
file_name	Имя файла для ввода.	char	размер 256	256

2.8 Функция insert:

Название переменной	Описание	Тип	Диапазон(х86)	Ёмкость(байт)
ptr	Новый созданный узел.	Node*	0x00000000 до 0xFFFFFFFF	4/8
right	Узел после вставки.	Node*	0x00000000 до 0xFFFFFFFF	4/8
left	Узел перед вставкой.	Node*	0x00000000 до 0xFFFFFFFF	4/8

2.9 Функция max_heap:

Название переменной	Описание	Тип	Диапазон(х86)	Ёмкость(байт)
ptr1	Текущий узел.	Node*	0x00000000 до 0xFFFFFFFF	4/8
ptr2	Левый потомок.	Node*	0x00000000 до 0xFFFFFFFF	4/8
ptr3	Правый потомок.	Node*	0x00000000 до 0xFFFFFFFF	4/8
value	Текущее значение узла	float	- 3.4E+38 до +3.4E+38	4

temp	Временное значение.	float	- 3.4E+38 до +3.4E+38	4
pre_value	Значение потомка.	float	- 3.4E+38 до +3.4E+38	4
check_heap	Флаг проверки кучи.	bool	False;True	4
value_leaf	Исходное значение узла.	float	- 3.4E+38 до +3.4E+38	4
i	Индекс текущего узла.	int	0 до $2^{31} - 1$	4

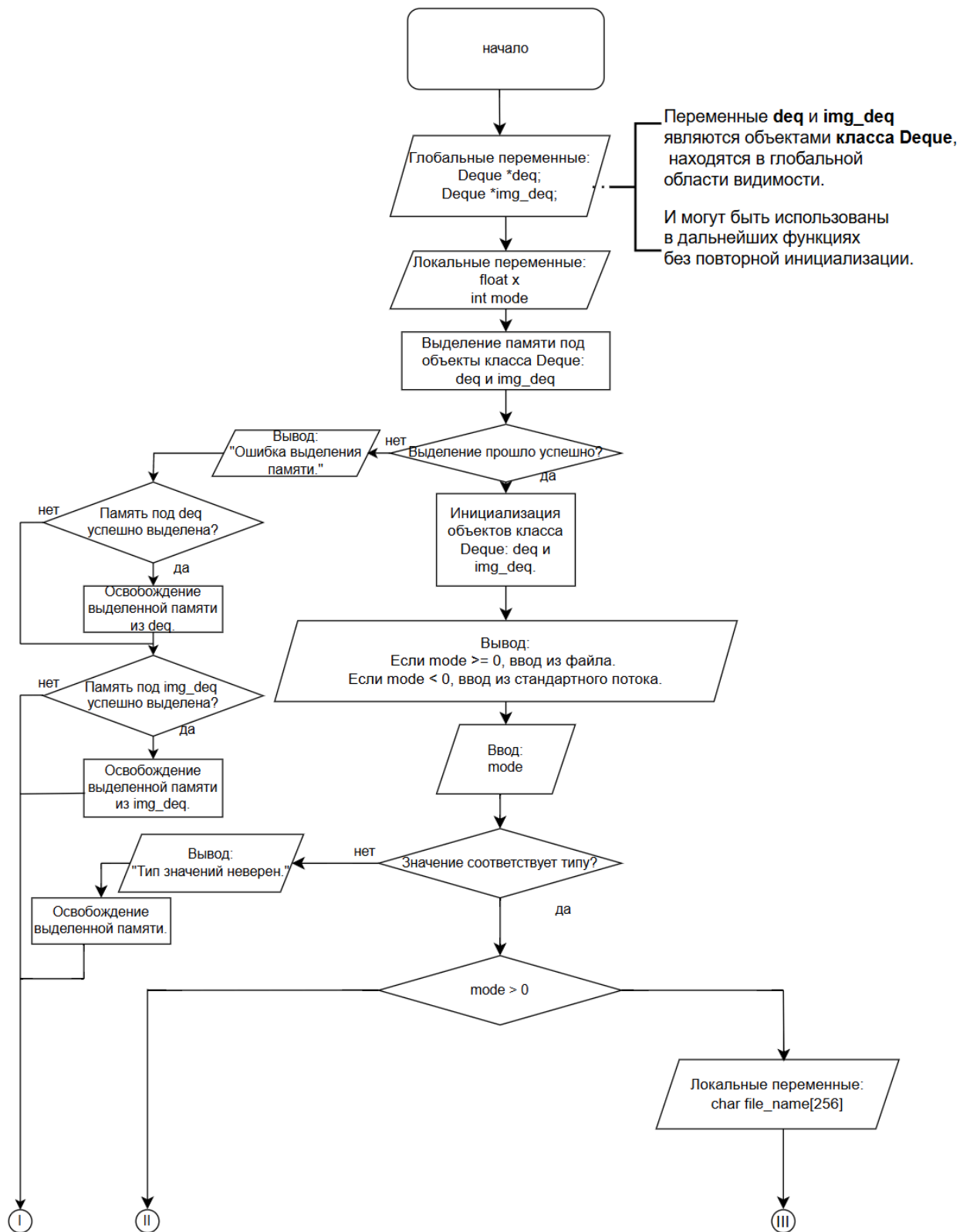
2.10 Функция pyramid_sort:

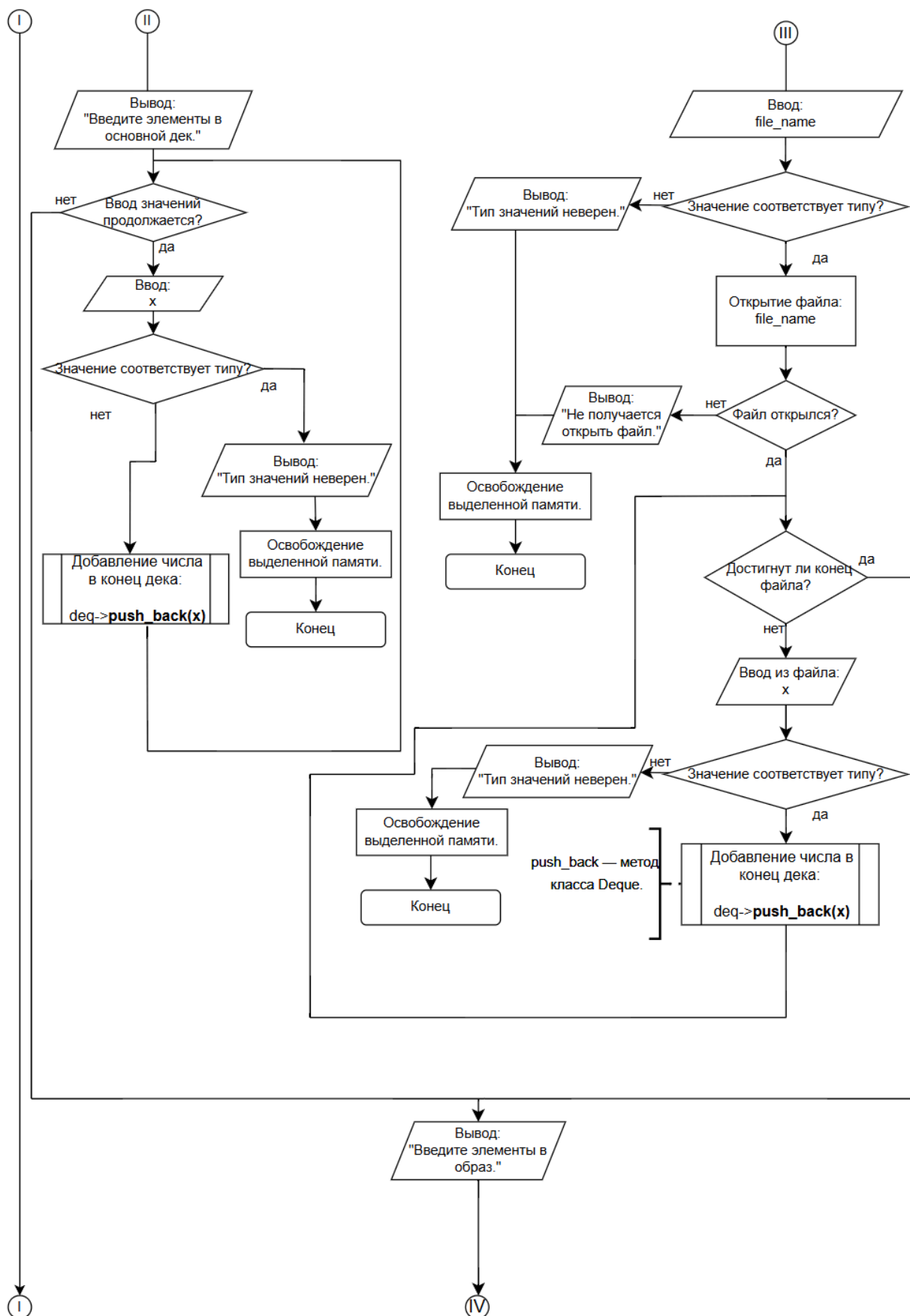
Название переменной	Описание	Тип	Диапазон(х86)	Ёмкость(байт)
ptr_tmp	Временный узел для обмена.	Node*	0x00000000 до 0xFFFFFFFF	4/8
n_size	Текущий размер неотсортированной части.	int	0 до $2^{31} - 1$	4
ptr	Последний элемент дека.	Node*	0x00000000 до 0xFFFFFFFF	4/8

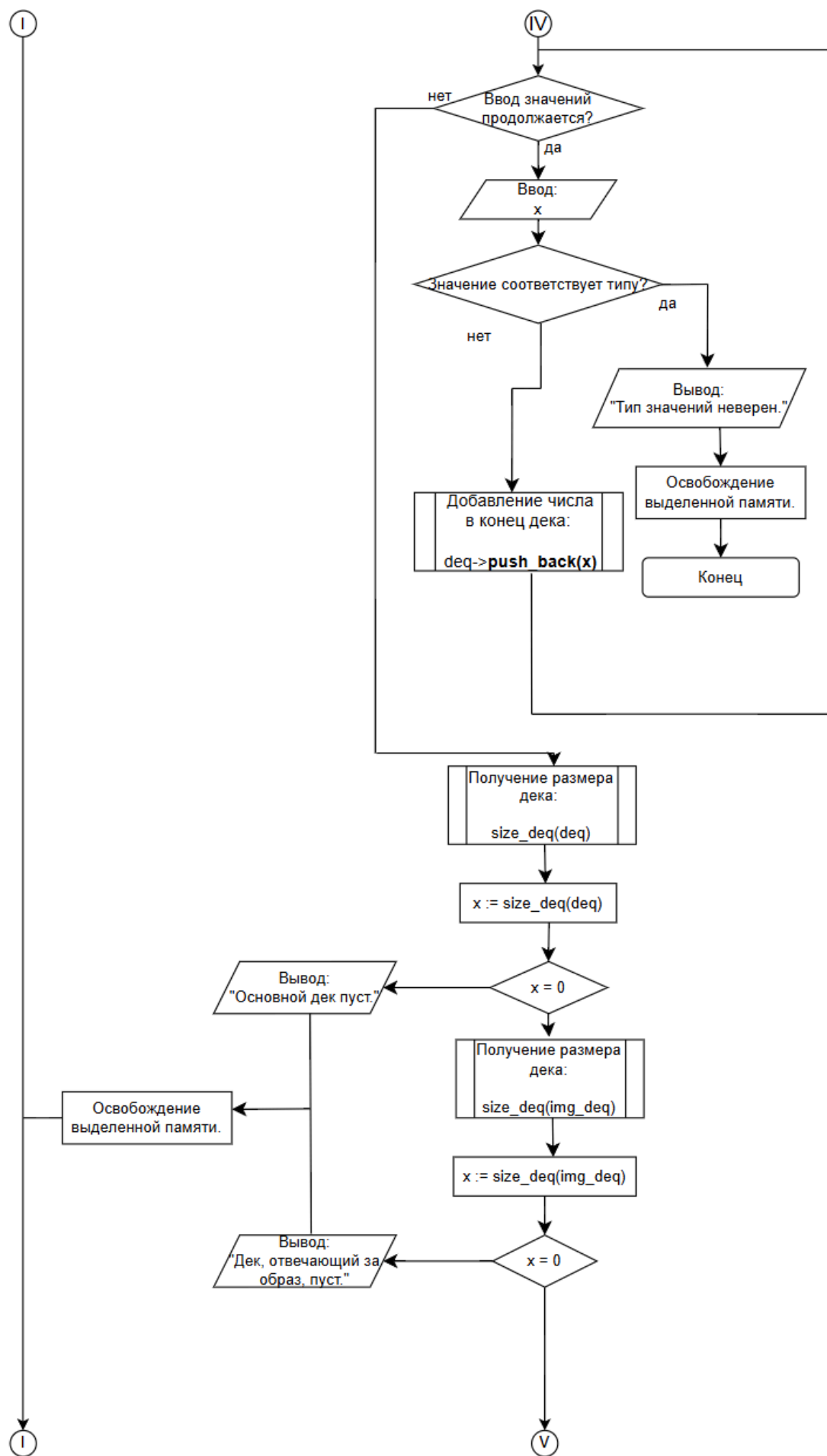
2.11 Функция KMP_SEQ:

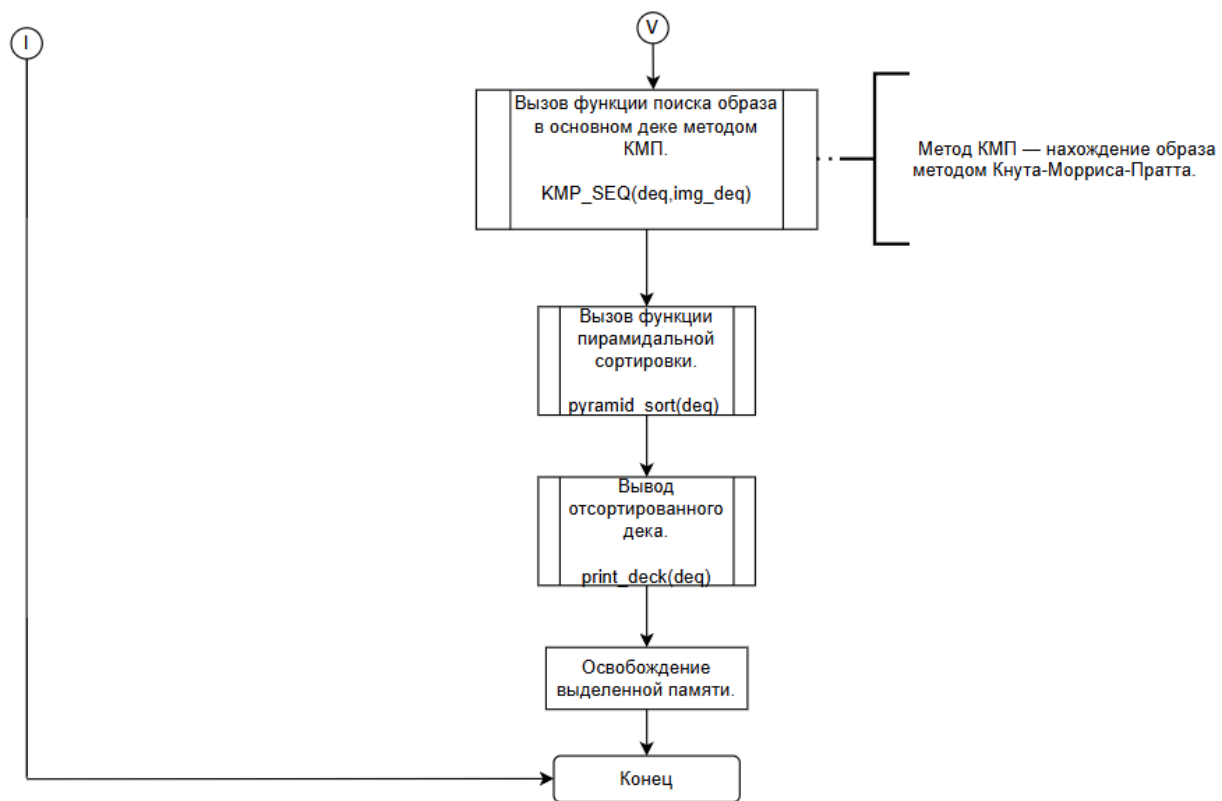
Название переменной	Описание	Тип	Диапазон(х86)	Ёмкость(байт)
ptr1	Узел последовательности.	Node*	0x00000000 до 0xFFFFFFFF	4/8
ptr2	Узел образца.	Node*	0x00000000 до 0xFFFFFFFF	4/8
ptr	Префикс-функция.	Deque*	0x00000000 до 0xFFFFFFFF	4/8
m	Длина образа.	int	0 до $2^{31} - 1$	4
n	Длина последовательности.	int	0 до $2^{31} - 1$	4
i,j	Индексы сравнения.	int	0 до $2^{31} - 1$	4

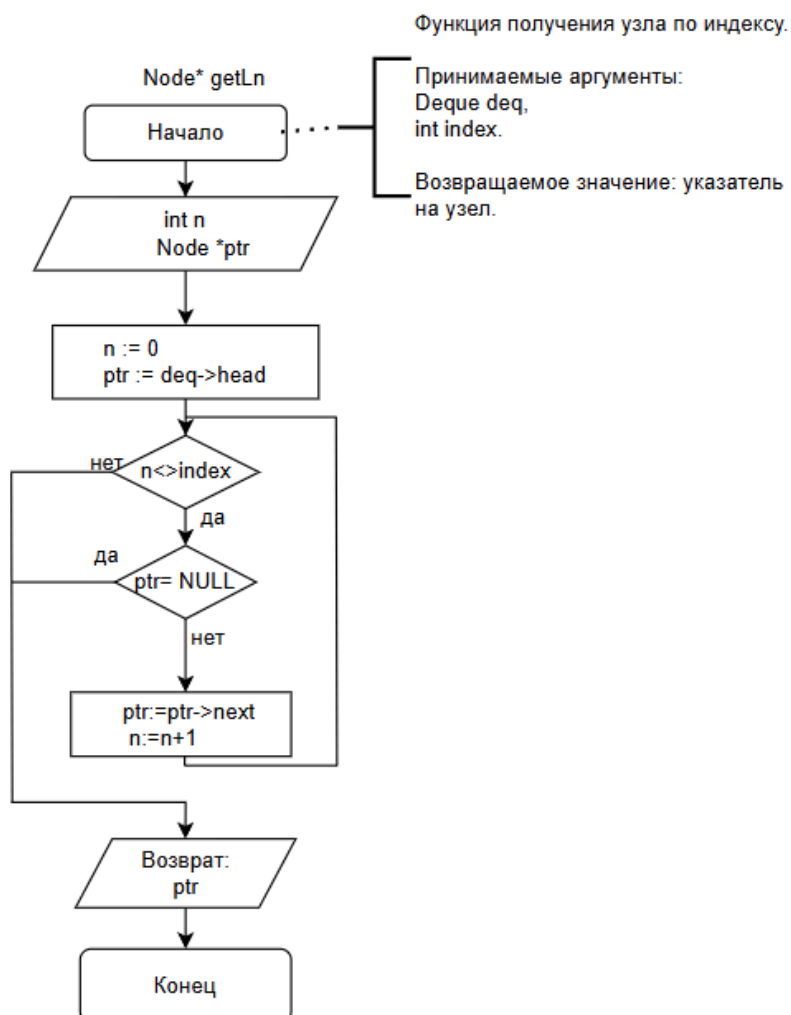
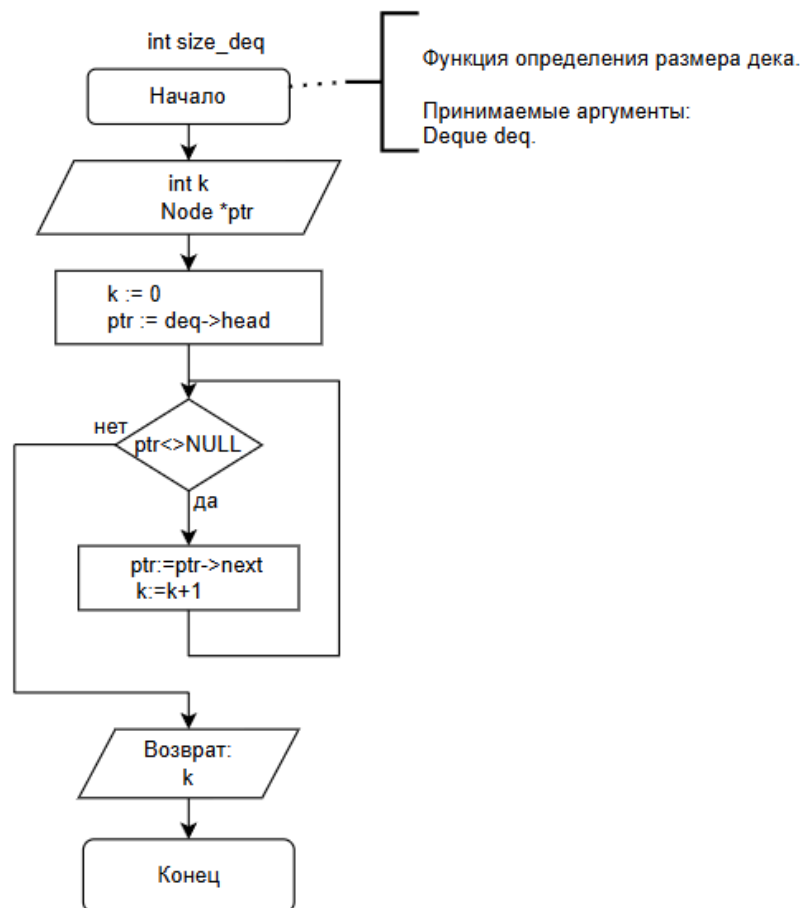
3.Блок-схема программы.

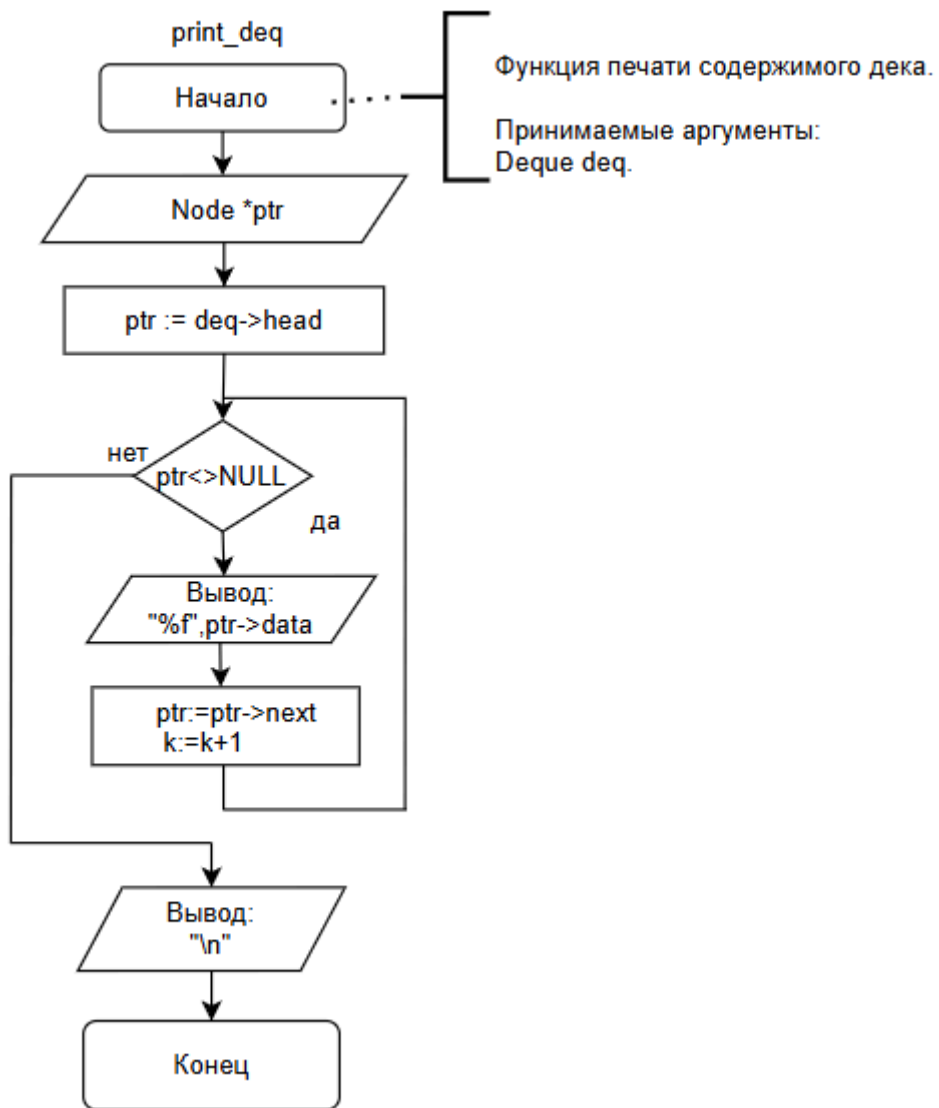


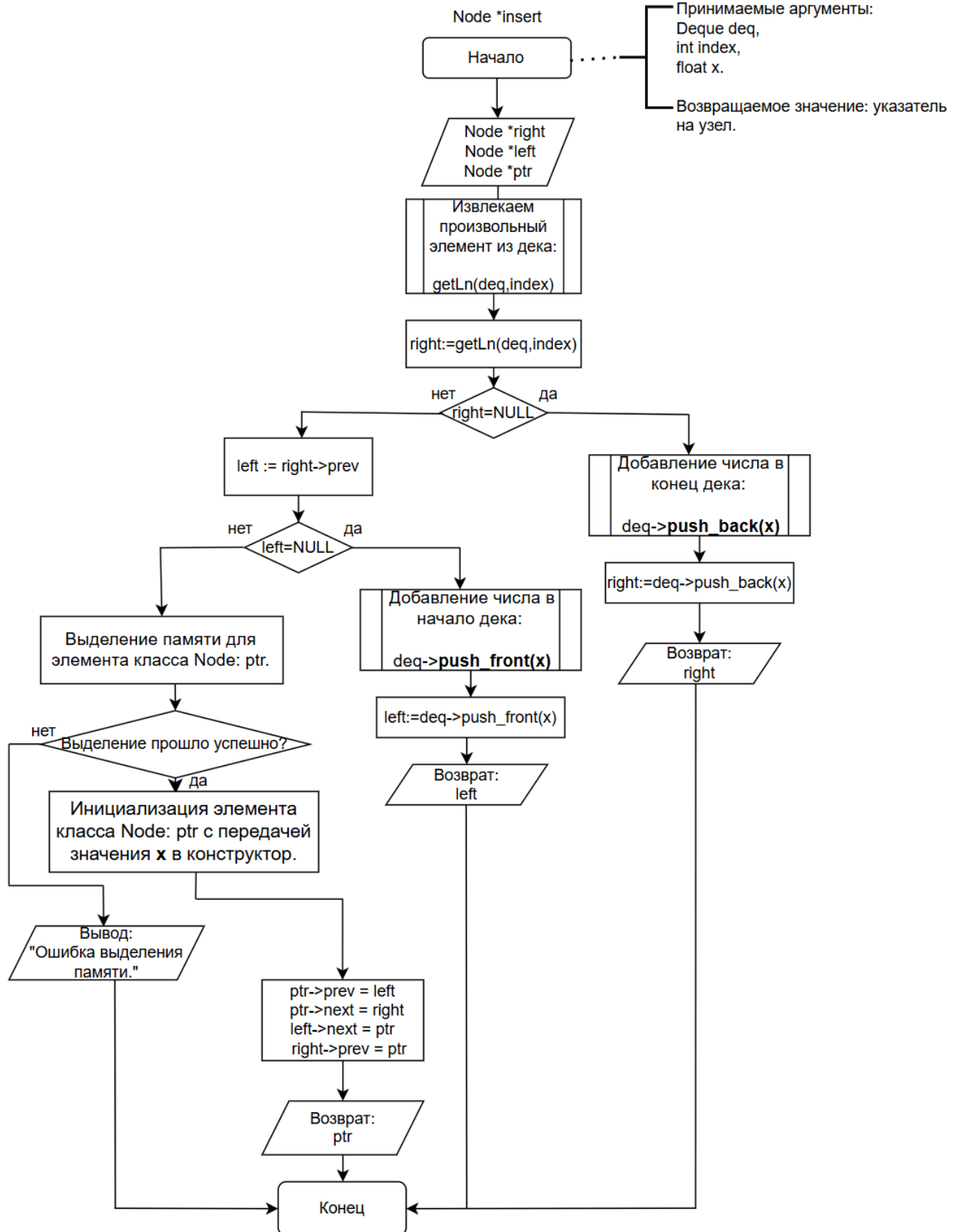


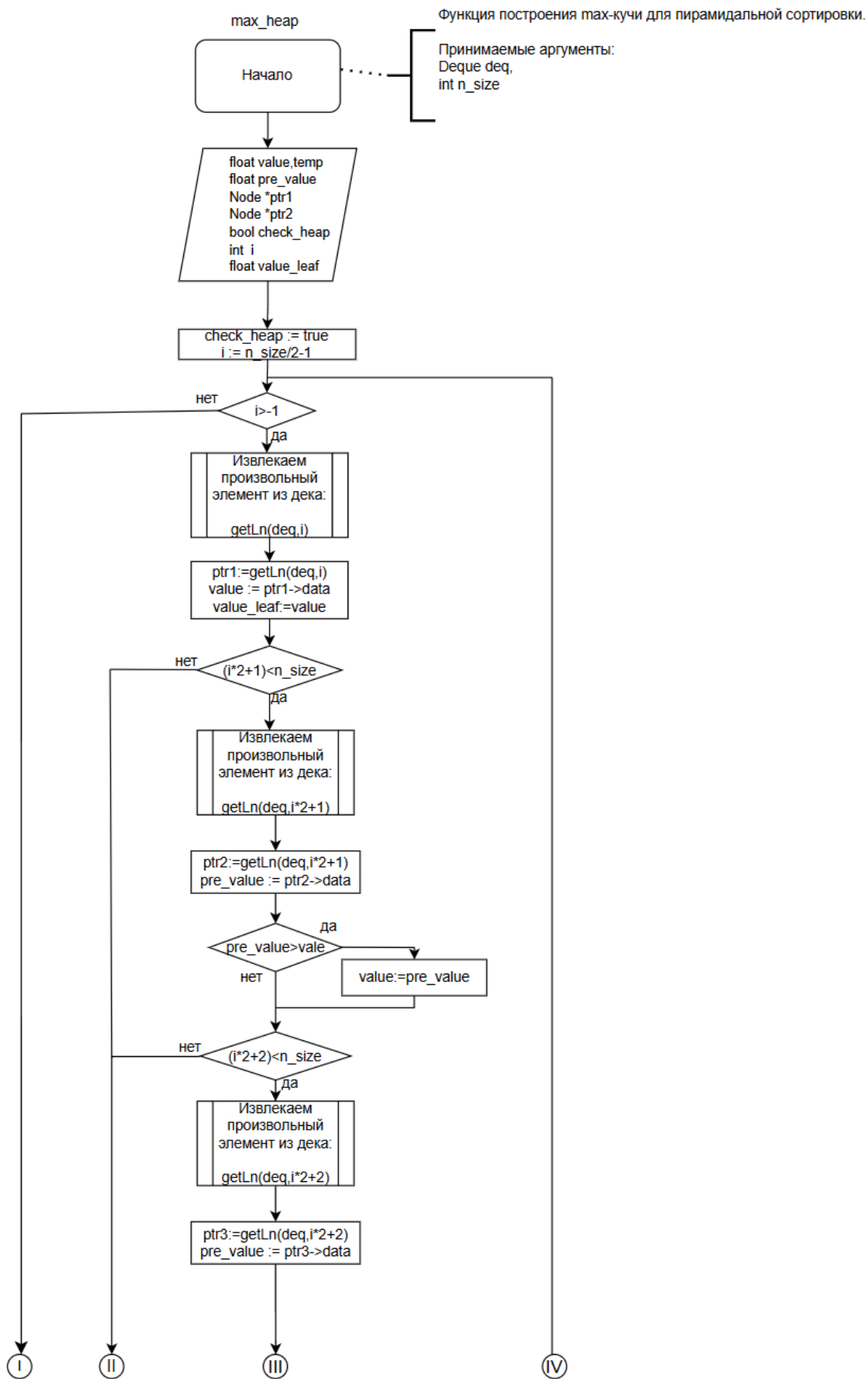


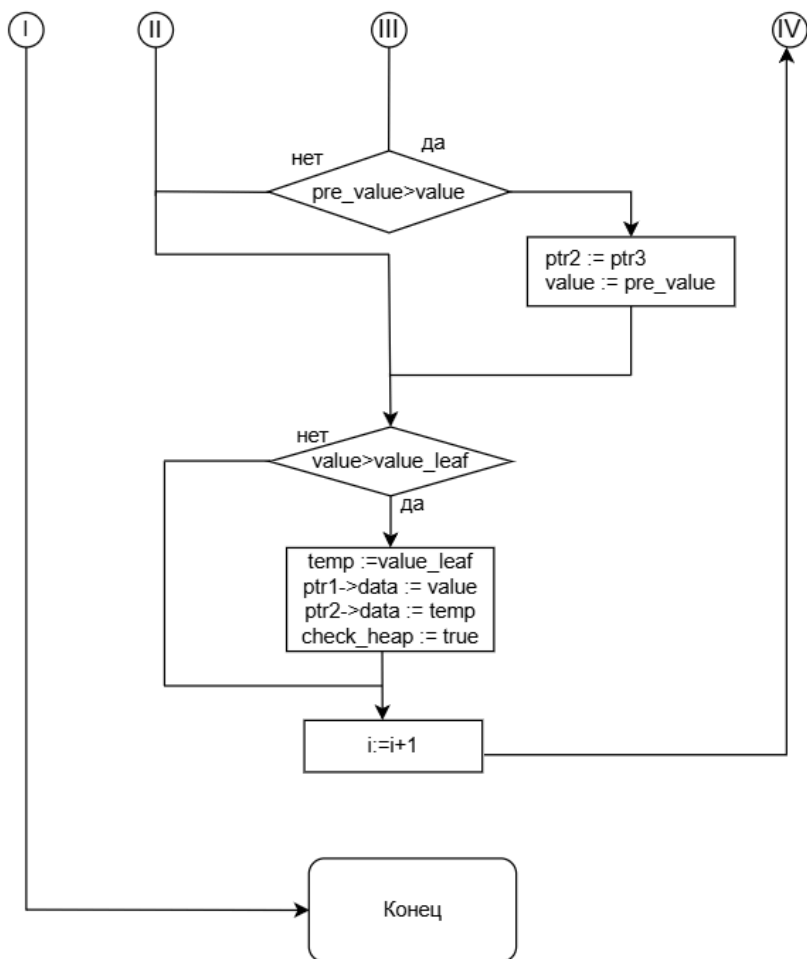


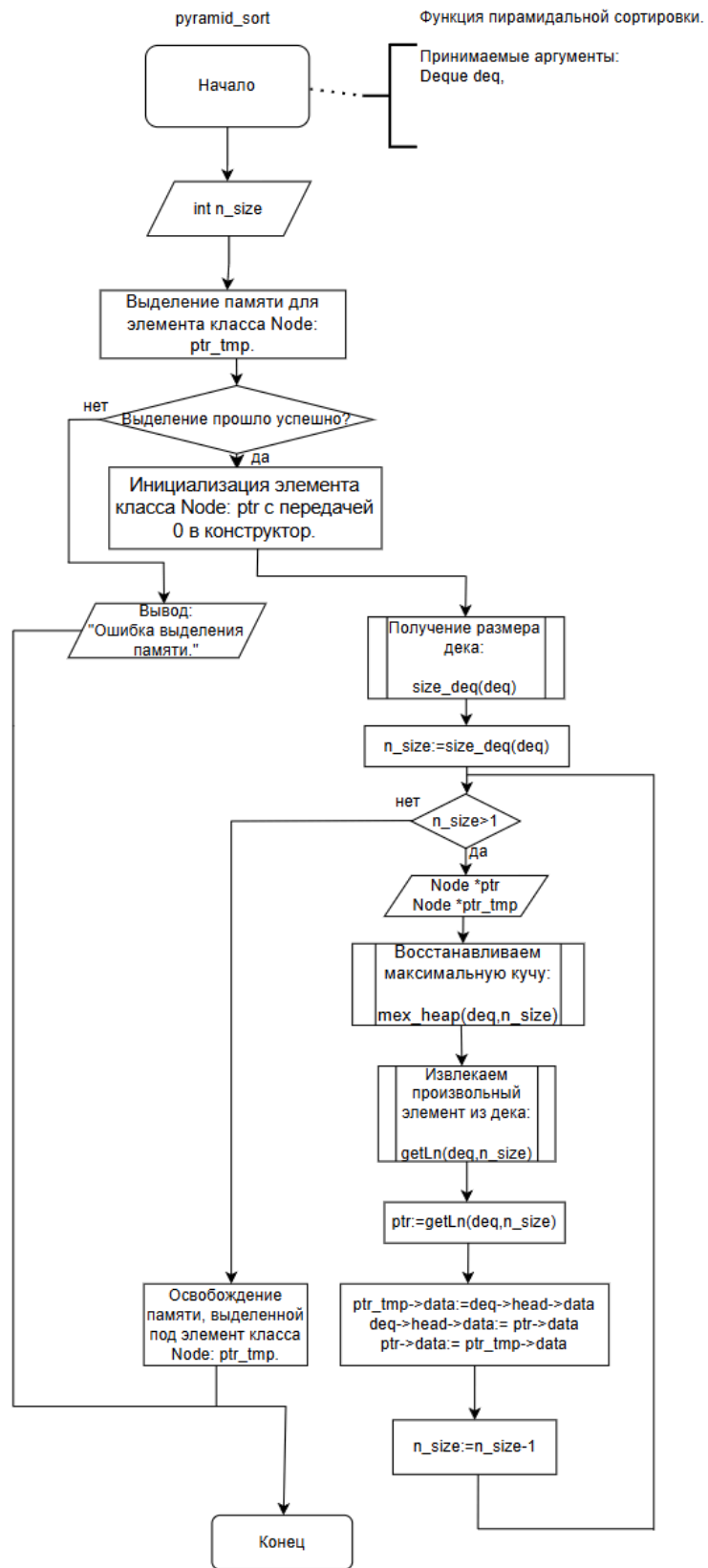


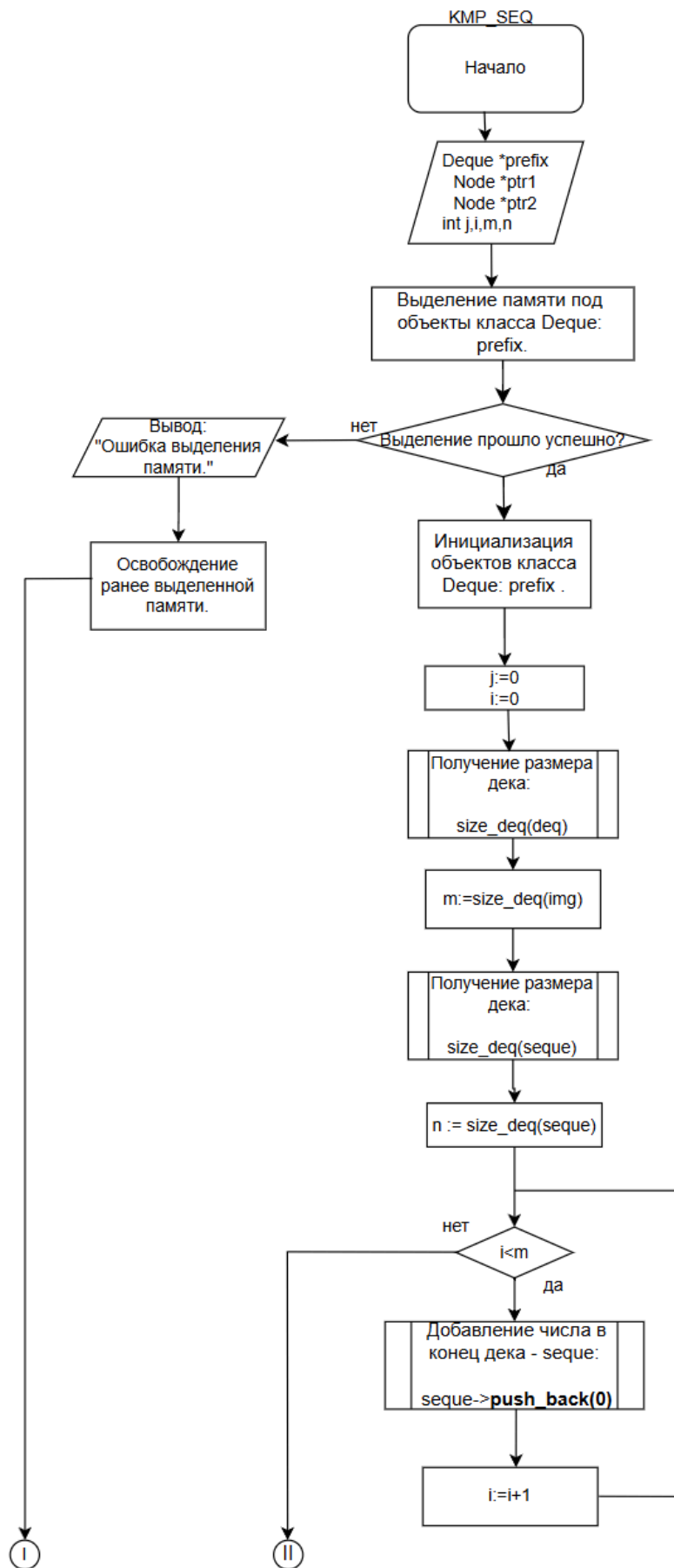


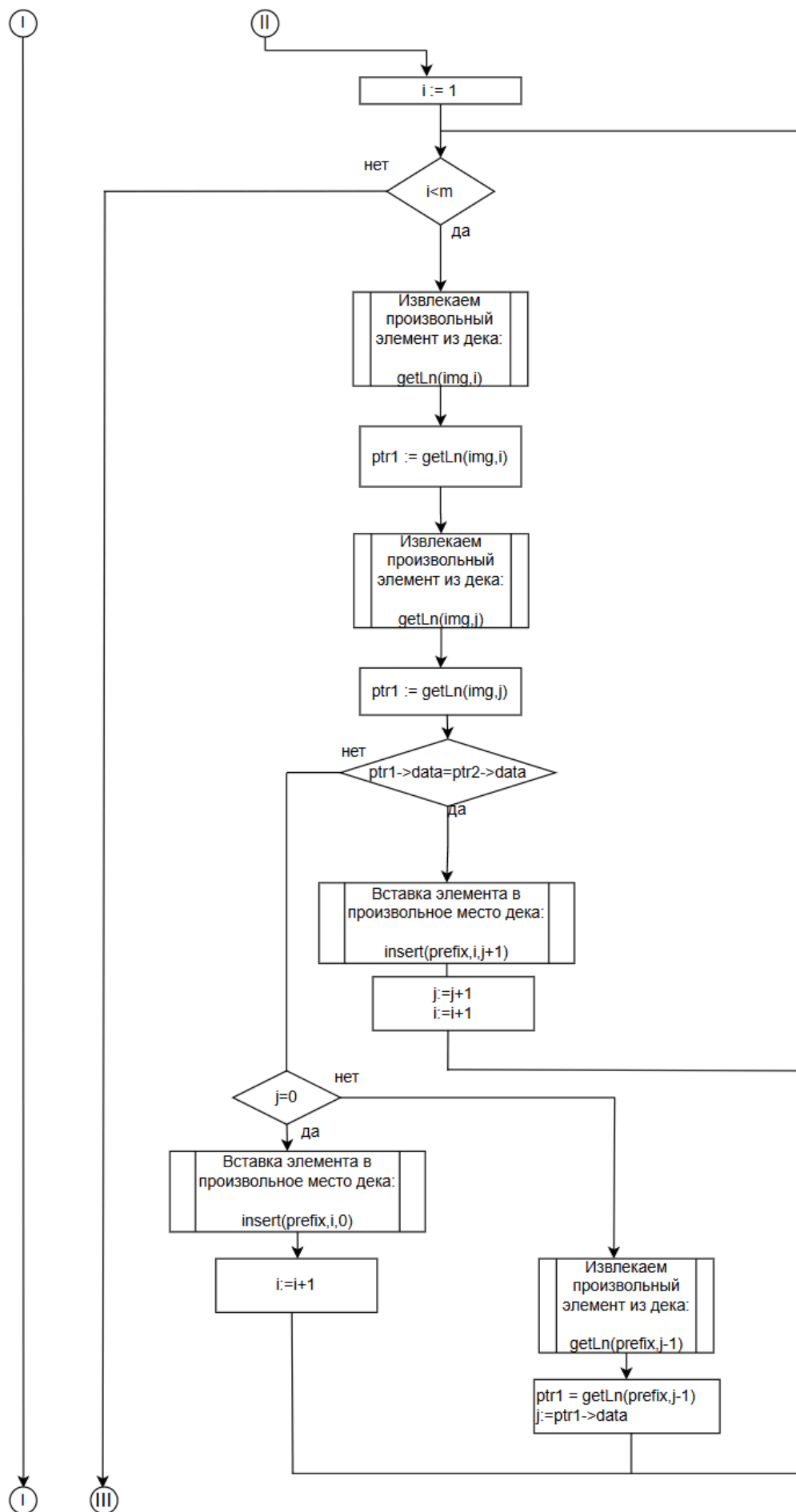


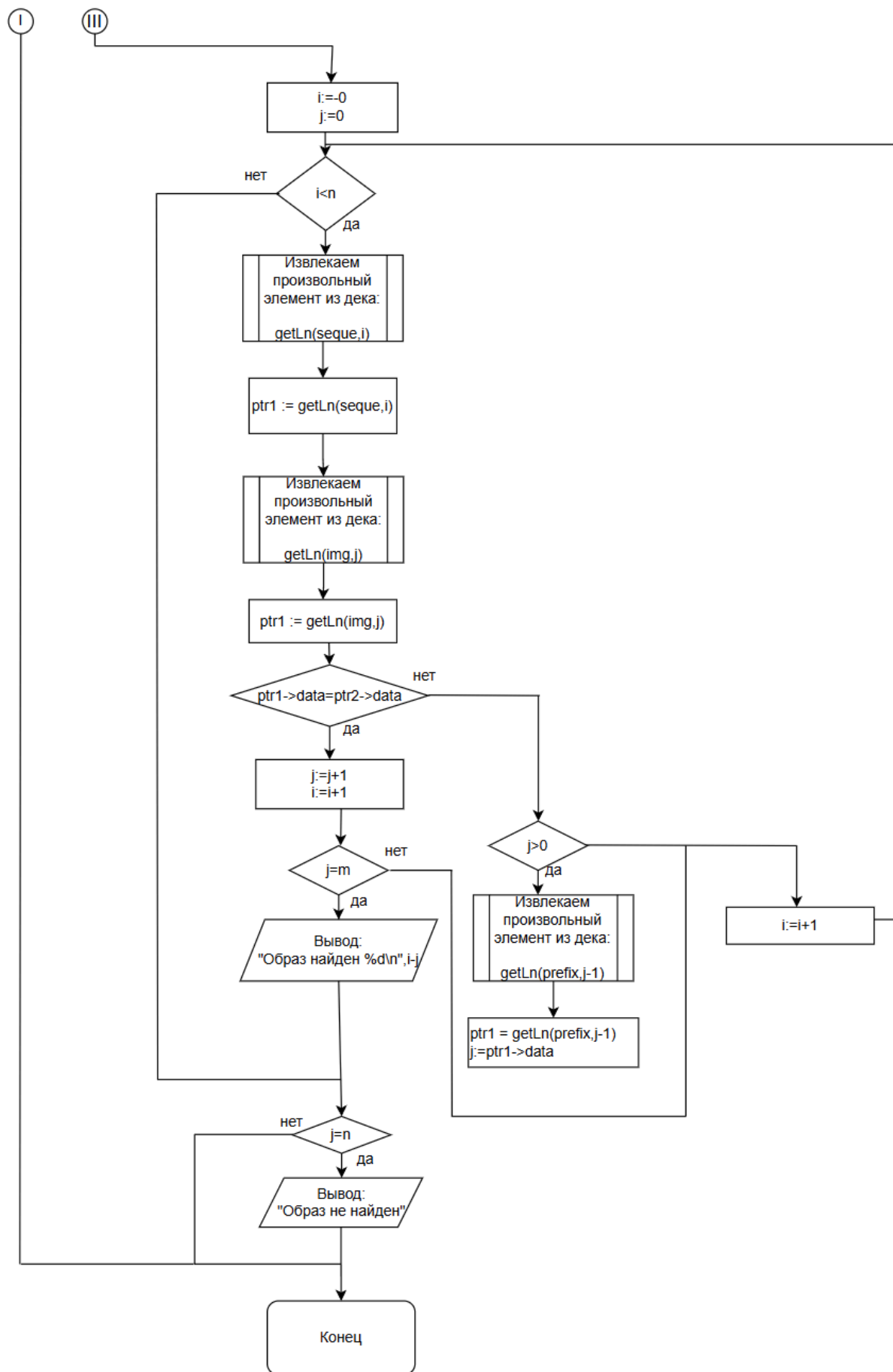




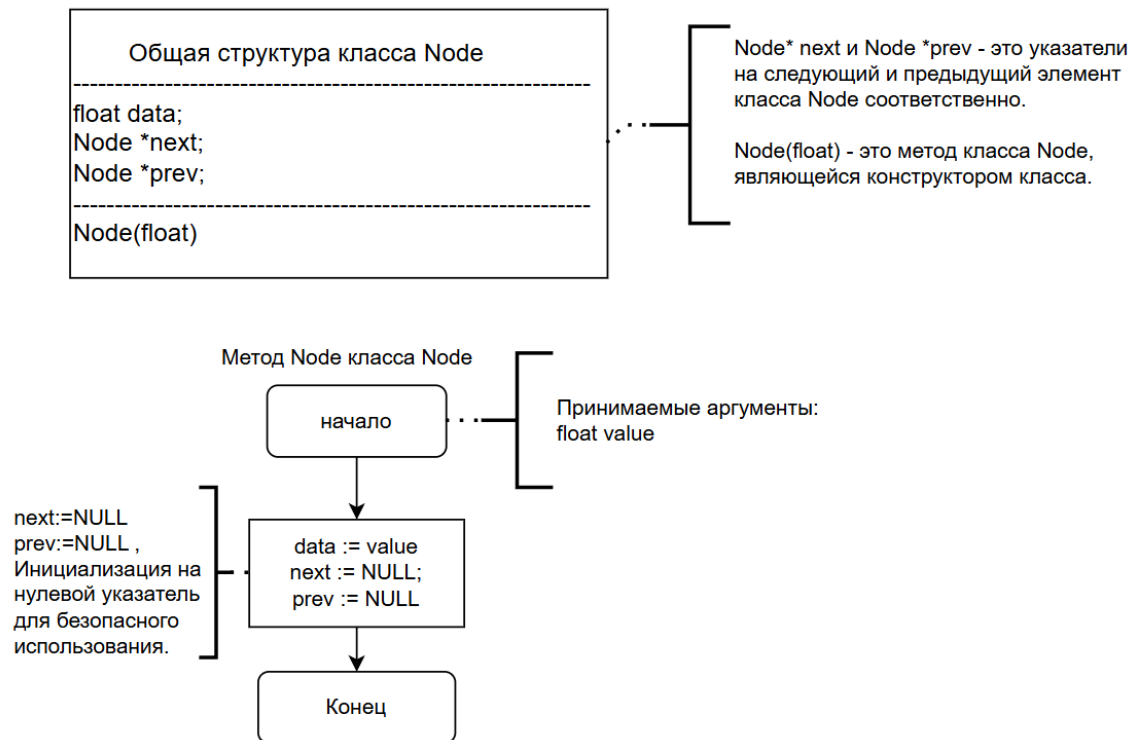




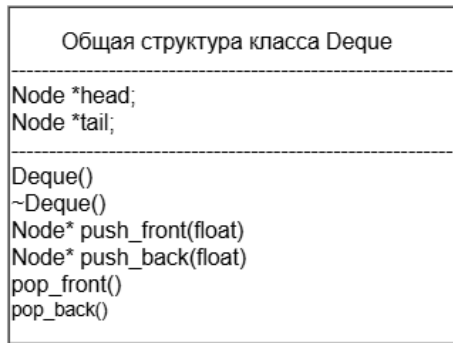




3.1 Блок-схема класса Node:



3.2 Блок-схема класса Deque:



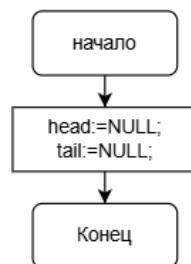
Node *head и Node *tail — это указатели на первый и последний элемент объекта класса Deque.

Deque() — это метод класса Deque, являющийся конструктором класса.

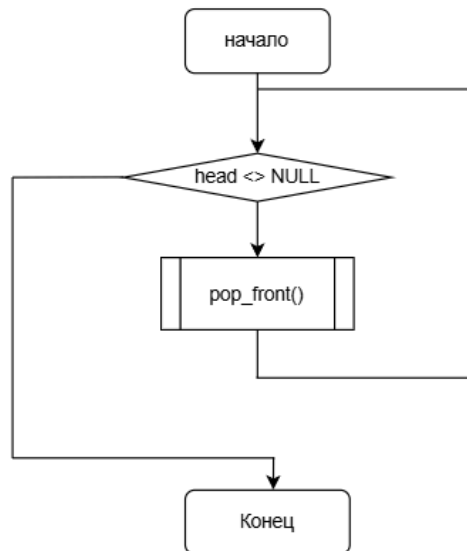
~Deque() — это метод класса Deque, являющийся деструктором класса.

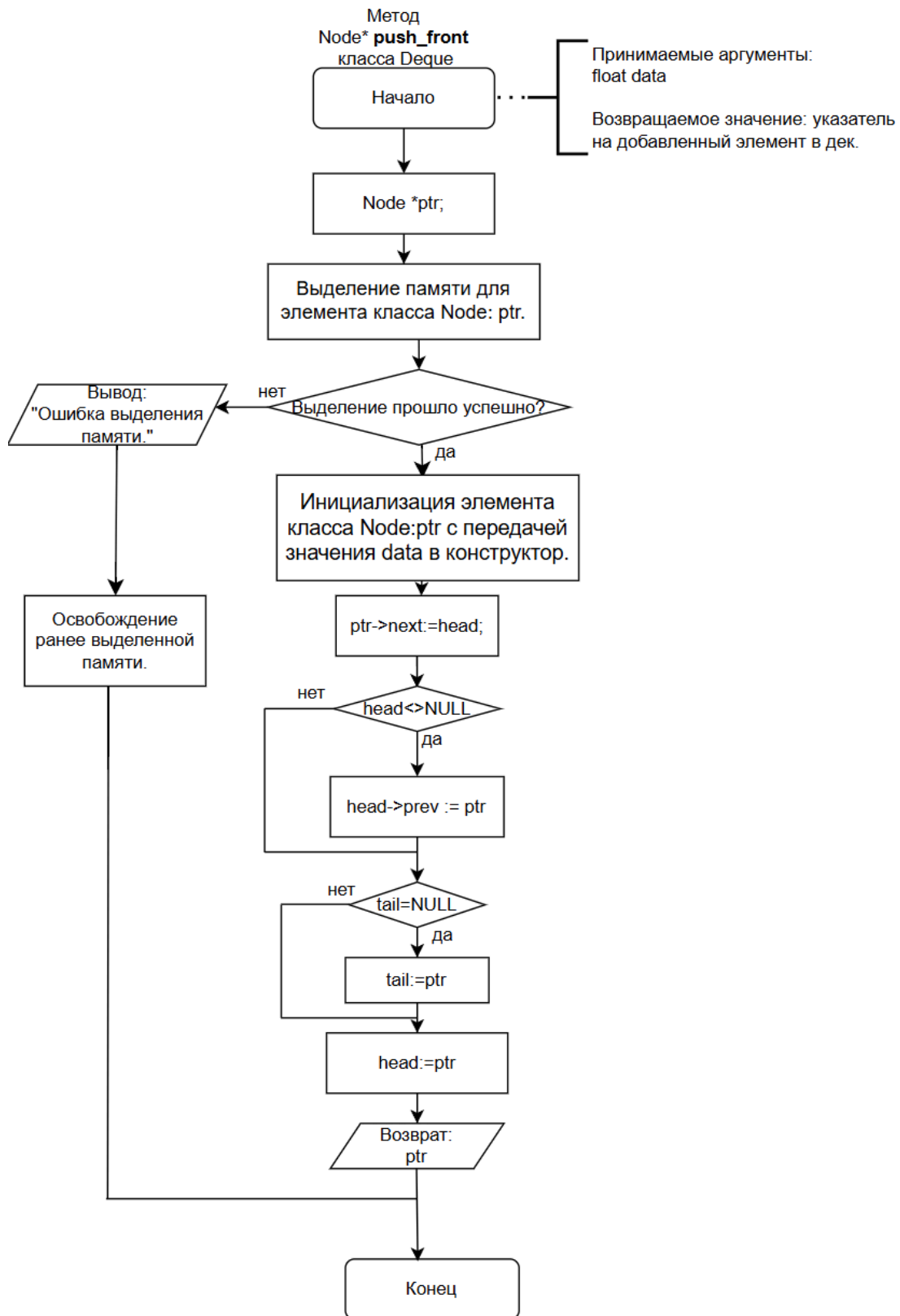
push_front(float), push_back(float), pop_front(), pop_back() — это методы класса, реализующие взаимодействие с объектом.

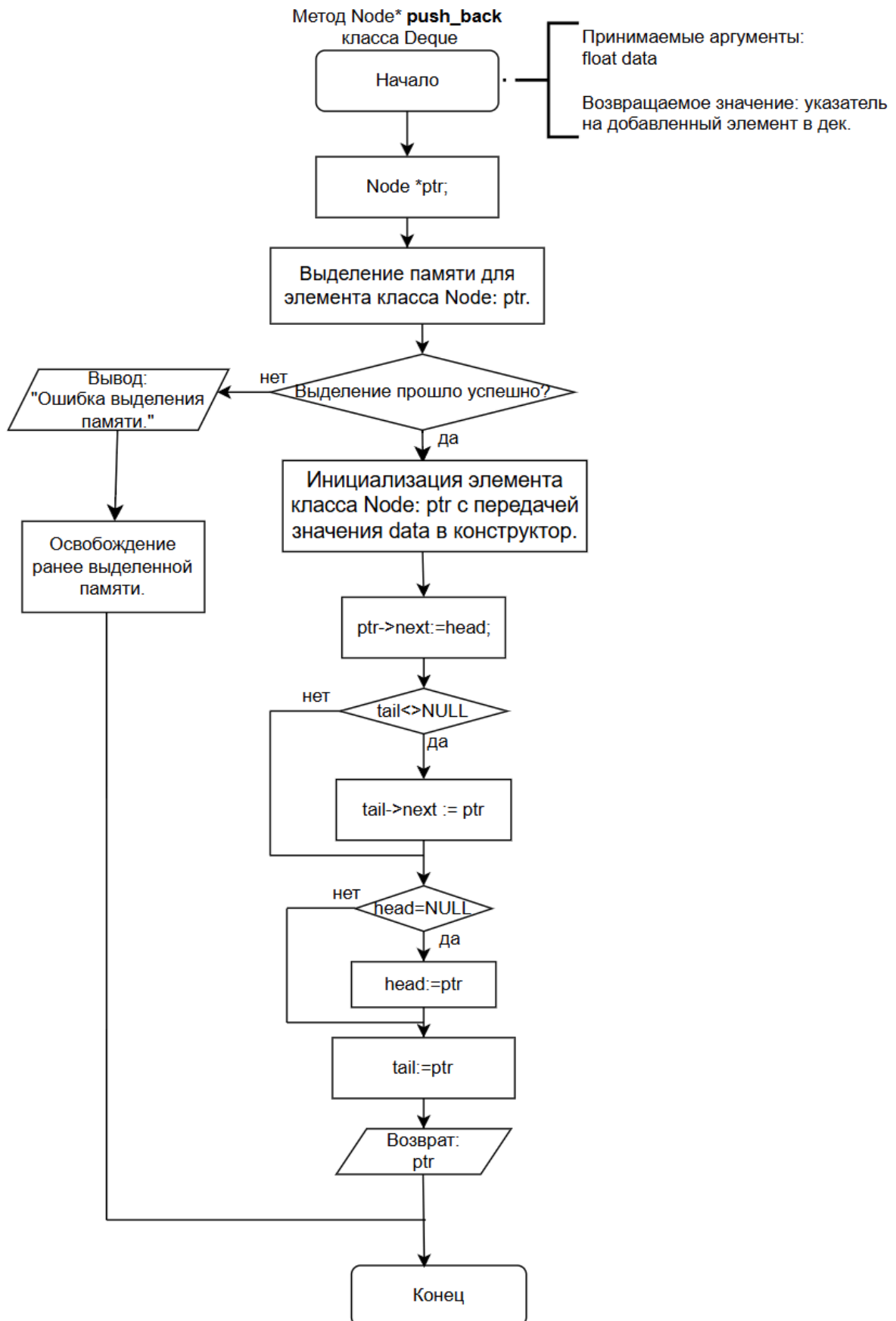
Метод Deque() класса Deque



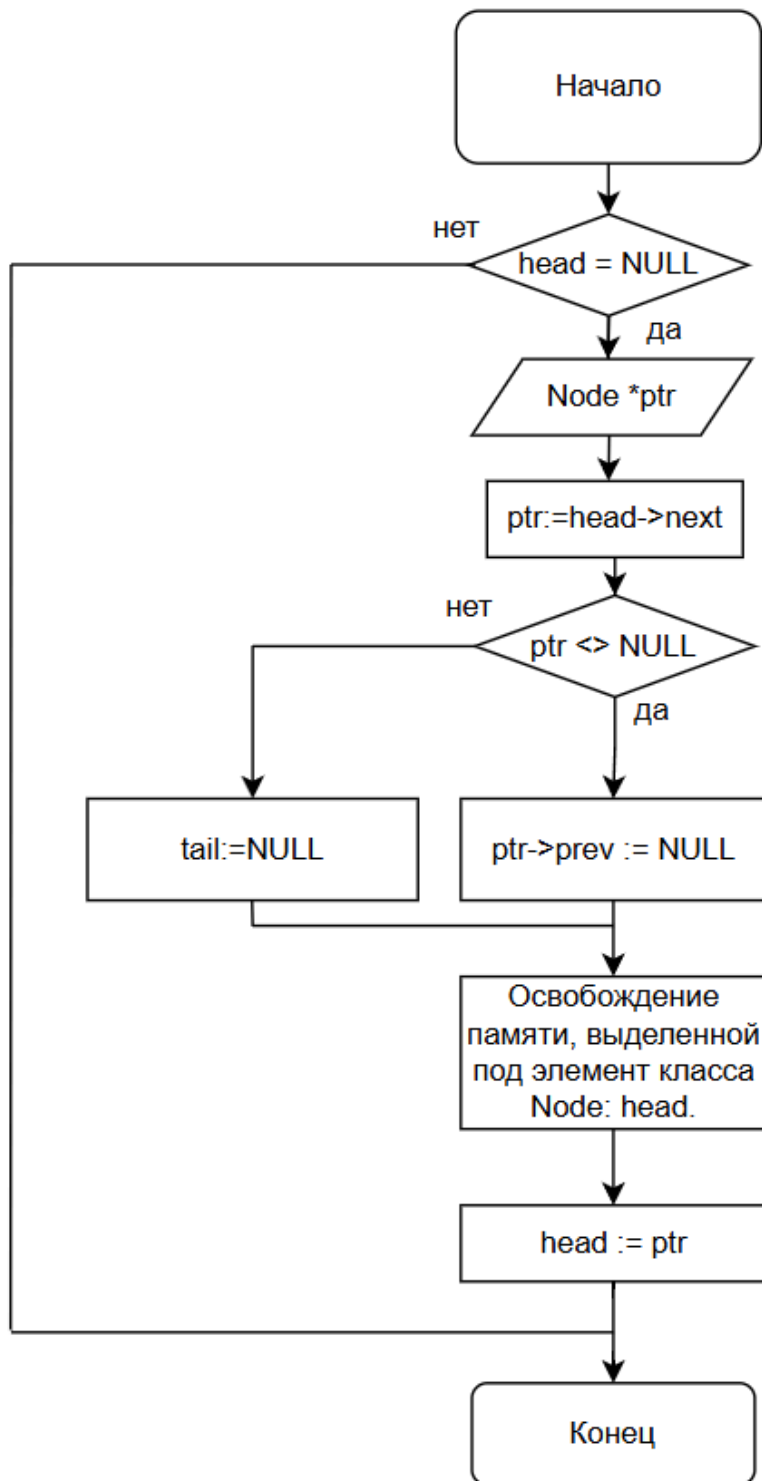
Метод ~Deque() класса Deque



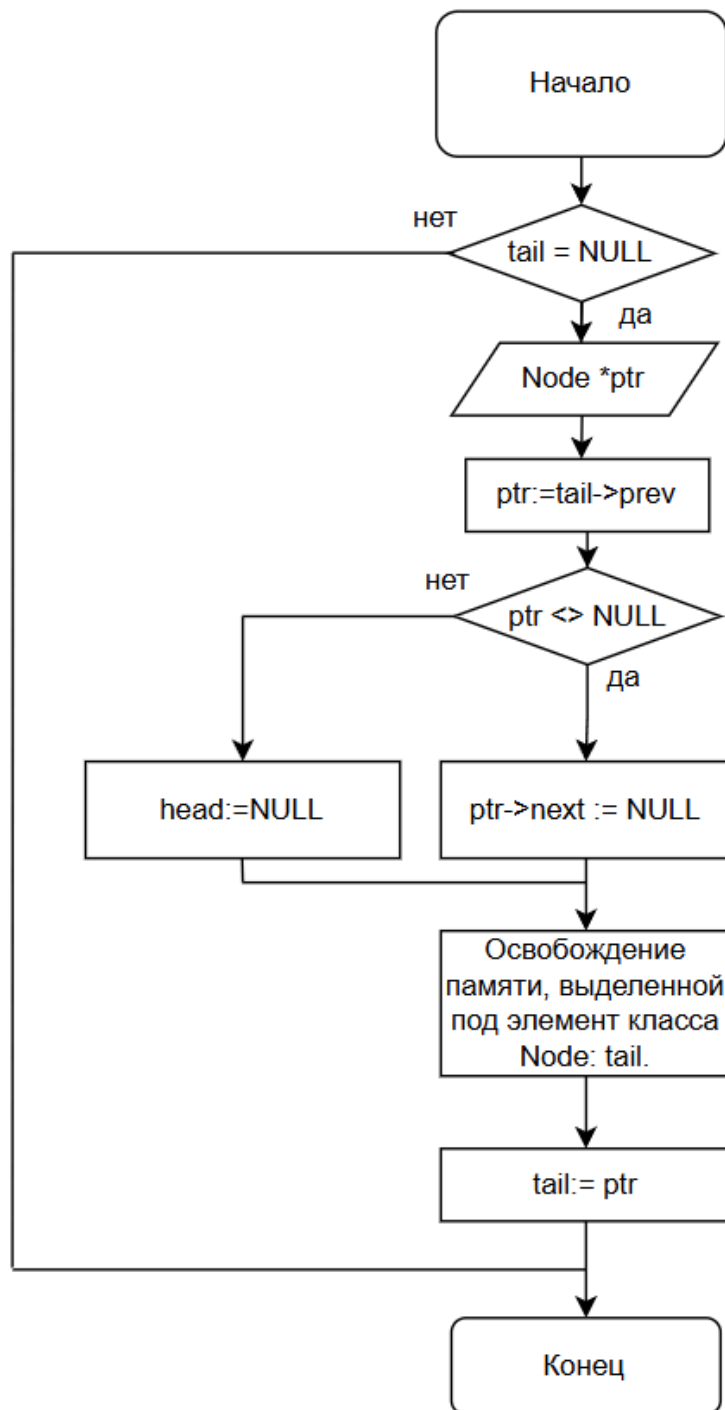




Метод **pop_front** класса Deque



Метод **pop_back** класса Deque



4. Код программы

```
#include <iostream>

#include <fstream>

// Объявление класса Deque (двусторонняя очередь)

class Deque;

// Глобальные указатели на основной дек и дек-образец

Deque *deq = nullptr;      // Основной дек

Deque *img_deq = nullptr;  // Дек-образец для поиска

// Класс узла дека

class Node {

public:

    float data;    // Хранимое значение

    Node* next;    // Указатель на следующий узел

    Node* prev;    // Указатель на предыдущий узел

public:

    // Конструктор узла

    Node(float data) {

        this->data = data;

        this->next = this->prev = NULL; // Инициализация указателей

    }

};

// Класс двусторонней очереди (дека)
```

```

class Deque {

public:

    Node *head; // Указатель на начало дека

    Node *tail; // Указатель на конец дека


public:

    // Конструктор - инициализация пустого дека

    Deque() {

        this->head = NULL;

        this->tail = NULL;

    }


    // Деструктор - освобождение памяти

    ~Deque() {

        // Последовательно удаляем все элементы с начала

        while (head != NULL) {

            pop_front();

        }

    }


    // Добавление элемента в начало дека

    Node* push_front(float data) {

        // Выделение памяти под новый узел

        Node *ptr = new(std::nothrow) Node(data);

        // Проверка выделения памяти

        if (ptr == nullptr) {

            std::cout << "Error of Memory\n";

            delete deq;

```

```

        delete img_deq;

        exit(EXIT_FAILURE);
    }

    // Настройка связей между узлами

    ptr->next = head;

    if (head != NULL)

        head->prev = ptr;

    if (tail == NULL)    // Если дек был пуст

        tail = ptr;

    head = ptr;    // Обновление головы дека

    return ptr;
}

// Добавление элемента в конец дека

Node* push_back(float data) {

    // Выделение памяти под новый узел

    Node *ptr = new(std::nothrow) Node(data);

    // Проверка выделения памяти

    if (ptr == nullptr) {

        std::cout << "Error of Memory\n";

        delete deq;

        delete img_deq;

        exit(EXIT_FAILURE);
    }

    // Настройка связей между узлами

    ptr->prev = tail;

```

```

        if (tail != NULL)

            tail->next = ptr;

        if (head == NULL)    // Если дек был пуста

            head = ptr;

        tail = ptr;    // Обновление хвоста дека

        return ptr;
    }

// Удаление элемента из начала дека

void pop_front() {

    if (head == NULL) return;    // Если дек был пуст

    Node* ptr = head->next;

    if (ptr != NULL)

        ptr->prev = NULL;

    else

        tail = NULL;    // Дек стал пустым

    delete head;    // Освобождение памяти

    head = ptr;    // Обновление головы дека

}

// Удаление элемента из конца дека

void pop_back() {

    if (tail == NULL) return;    // Если дек был пуст

    Node *ptr = tail->prev;

    if (ptr != NULL)

        ptr->next = NULL;

    else

```



```

        head = NULL;    // Дек стал пустым

        delete tail;    // Освобождение памяти

        tail = ptr;    // Обновление хвоста дека
    }

};

// функция определения размера дека

int size_deq(Deque *deq) {

    int k = 0;

    Node *ptr = deq->head;

    // Проход по всем элементам дека

    while (ptr != NULL) {

        ptr = ptr->next;

        k++;    // Подсчет элементов

    }

    return k;

}

// функция получения узла по индексу

Node* getLn(Deque *deq, int index) {

    Node* ptr = deq->head;

    int n = 0;

    // Поиск узла с нужным индексом

    while (n != index) {

        if (ptr == NULL)

            return ptr;    // Если индекс превышает размер

        ptr = ptr->next;
    }
}

```

```

        n++;

    }

    return ptr;
}

// Функция печати содержимого дека

void print_deck(Deque *deq) {

    Node *ptr = deq->head;

    // Последовательный вывод всех элементов

    while (ptr != NULL) {

        printf("%0.2f ", ptr->data);

        ptr = ptr->next;

    }

    printf("\n");

}

// Функция вставки элемента по индексу

Node* insert(Deque *deq, int index, double x) {

    Node* right = getLn(deq, index);

    if (right == NULL)    // Если индекс в конце

        return deq->push_back(x);

    Node* left = right->prev;

    if (left == NULL)    // Если индекс в начале

        return deq->push_front(x);

    // Создание нового узла

    Node* ptr = new(std::nothrow) Node(x);

    if (ptr == nullptr) {

```

```

        std::cout << "Error of Memory\n";

        delete deq;

        delete img_deq;

        exit(EXIT_FAILURE);

    }

    // Вставка между left и right

    ptr->prev = left;

    ptr->next = right;

    left->next = ptr;

    right->prev = ptr;

    return ptr;

}

// Функция построения max-кучи для пирамидальной сортировки

void max_heap(Deque *deq, int n_size) {

    float value, temp;

    float pre_value;

    Node *ptr1 = nullptr;

    Node *ptr2 = nullptr;

    Node *ptr3 = nullptr;

    bool check_heap = true;

    // Пока куча не упорядочена

    while (check_heap == true) {

        check_heap = false;

        float value_leaf;

```

```

// Проход по всем родительским узлам

for (int i = n_size / 2 - 1; i > -1; i--) {

    ptr1 = getLn(deq, i);

    value = ptr1->data;

    value_leaf = value;

    // Проверка левого потомка

    if (i * 2 + 1 < n_size) {

        ptr2 = getLn(deq, i * 2 + 1);

        pre_value = ptr2->data;

        if (pre_value > value) {

            value = pre_value;

        }

        // Проверка правого потомка

        if (i * 2 + 2 < n_size) {

            ptr3 = getLn(deq, i * 2 + 2);

            pre_value = ptr3->data;

            if (pre_value > value) {

                ptr2 = ptr3;

                value = pre_value;

            }

        }

    }

    // Если нашли большее значение - меняем местами

    if (value > value_leaf) {

```

```

        temp = value_leaf;

        ptr1->data = value;

        ptr2->data = temp;

        check_heap = true; // Требуется еще проход
    }

}

}

}

// функция пирамидальной сортировки

void pyramid_sort(Deque *deq) {

    // Временный узел для обмена значений

    Node *ptr_tmp = new(std::nothrow) Node(0);

    if (ptr_tmp == nullptr) {

        std::cout << "Error of Memory!" << std::endl;

        delete deq;

        delete img_deq;

        exit(EXIT_FAILURE);

    }

    int n_size = size_deq(deq);

    // Основной цикл сортировки

    while (n_size > 1) {

        max_heap(deq, n_size); // Построение кучи

        Node *ptr = getLn(deq, n_size - 1);

        // Обмен максимального элемента с последним

        ptr_tmp->data = deq->head->data;

        deq->head->data = ptr->data;

        ptr->data = ptr_tmp->data;
    }
}

```

```

        n_size--; // Уменьшение размера неотсортированной части
    }

    delete ptr_tmp; // Освобождение временного узла
}

// Функция поиска подпоследовательности (алгоритм Кнута-Морриса-Пратта)
void KMP_SEQ(Deque *seque, Deque *img) {

    Node *ptr1 = nullptr;

    Node *ptr2 = nullptr;

    // Создание дека для хранения префикс-функции
    Deque *prefix = new(std::nothrow) Deque();

    if (prefix == nullptr) {

        delete deq;

        delete img_deq;

        exit(EXIT_FAILURE);

    }

    int m = size_deq(img); // Длина образца

    int n = size_deq(seque); // Длина последовательности

    // Инициализация префикс-функции нулями
    for (int i = 0; i < m; i++)

        prefix->push_back(0);

    // Вычисление префикс-функции для образца

    int j = 0, i = 1;

    while (i < m) {

```

```

ptr1 = getLn(img, i);

ptr2 = getLn(img, j);

if (ptr1->data == ptr2->data) {

    insert(prefix, i, j + 1);

    j++;

    i++;

} else if (j == 0) {

    insert(prefix, i, 0);

    i++;

} else {

    ptr1 = getLn(prefix, j - 1);

    j = ptr1->data;

}

}

// Поиск образца в последовательности

i = 0;

j = 0;

while (i < n) {

    ptr1 = getLn(seque, i);

    ptr2 = getLn(img, j);

    if (ptr1->data == ptr2->data) {

        i += 1;

        j += 1;

        if (j == m) { // Образец полностью найден

            printf("image find %d\n", i - j); // Вывод позиции

            return;

```

```

    }

    } else if (j > 0) {

        ptr1 = getLn(prefix, j - 1);

        j = ptr1->data;  // Сдвиг по префикс-функции

    } else {

        i++;  // Сдвиг последовательности

    }

}

// Образец не найден

if (i == n)

    printf("image not find \n");

return;

}

int main(void) {

    // Инициализация деков

    deq = new(std::nothrow) Deque();  // Дек для данных

    img_deq = new(std::nothrow) Deque();  // Дек для образца

    // Проверка выделения памяти

    if (deq == nullptr || img_deq == nullptr) {

        std::cerr << "Error of Memory!" << std::endl;

        if (deq != nullptr)

            delete deq;

        if (img_deq != nullptr)

            delete img_deq;

        exit(EXIT_FAILURE);
    }
}

```



```

}

float x;

int mode;

// Выбор источника ввода данных

std::cout << "write from StdI <= 0, FILE > 0." << std::endl;

std::cin >> mode;

// Проверка корректности ввода

if (std::cin.fail()) {

    std::cerr << "Error: Bad value!" << std::endl;

    return 0;

}

// Ввод из файла

if (mode > 0) {

    char file_name[256];

    std::cin >> file_name;

    if (std::cin.fail()) {

        std::cerr << "Error: Bad value!" << std::endl;

    }

    std::ifstream ifs(file_name);

    if (ifs.is_open()) {

        // Чтение данных из файла

        while (ifs >> x) {

            if (ifs.fail()) {

                std::cerr << "Error: Bad value!" << std::endl;

```

```

        delete deq;

        delete img_deq;

        exit(EXIT_FAILURE);

    }

    deq->push_back(x); // Добавление в дек

}

} else {

    std::cerr << "Couldn't open the file!" << std::endl;

    delete deq;

    delete img_deq;

    exit(EXIT_FAILURE);

}

// Вывод введенных данных

std::cout << "Your previous deque:" << std::endl;

print_deck(deq);

} else {

    // Ввод данных с клавиатуры

    std::cout << "Enter the previos deque:" << std::endl;

    while (!feof(stdin)) {

        std::cin >> x;

        if (std::cin.fail() && !std::cin.eof()) {

            std::cerr << "Error: Bad value!" << std::endl;

            delete deq;

            delete img_deq;

            exit(EXIT_FAILURE);

        }

        if (!std::cin.eof())

            deq->push_back(x); // Добавление в дек

```

```

    }

}

// Ввод образца для поиска

std::cin.clear();

std::cout << "Enter the image:" << std::endl;

while (std::cin >> x) {

    if (std::cin.fail() && !std::cin.eof()) {

        std::cerr << "Error: Bad value!" << std::endl;

        delete deq;

        delete img_deq;

        exit(EXIT_FAILURE);

    }

    if (!std::cin.eof())

        img_deq->push_back(x); // Добавление в дек-образец

}

// Проверка на пустые деки

if (size_deq(deq) == 0) {

    fprintf(stderr, "Error: Previos deque is empty!");

    delete deq;

    delete img_deq;

    exit(EXIT_FAILURE);

} else if (size_deq(img_deq) == 0) {

    fprintf(stderr, "Error: Previos image-deque is empty!");

    delete deq;

    delete img_deq;

    exit(EXIT_FAILURE);

}

```

```

// Поиск образца в исходной и отсортированной последовательности

std::cout <<
"-----" << std::endl;

printf("seque in a previos deque: ");

KMP_SEQ(deq, img_deq); // Поиск в исходном deque

pyramid_sort(deq); // Сортировка deque

printf("seque in a sorted deque: ");

KMP_SEQ(deq, img_deq); // Поиск в отсортированном deque

print_deck(deq); // Вывод отсортированного deque

// Освобождение памяти

delete deq;

delete img_deq;

}

```

5. Программные тесты.

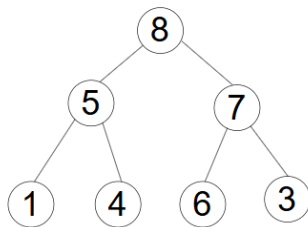
5.1:

Данные, которые подаются в программу со стандартного ввода:

[8, 5, 7, 1, 4, 6, 3] - данные для основного дека

[1] - образец

Представление данных основного дека в виде бинарного дерева:



Ожидаемый вывод:

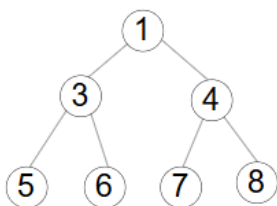
Расположение образца:

В исходном деке на 3 индексе.

В отсортированном на 0 индексе.

[1, 3, 4, 5, 6, 7, 8] - отсортированный дек.

Представление данных основного отсортированного дека в виде бинарного дерева:



Программный вывод:

```
write from StdI <= 0, FILE > 0.
```

```
-1
```

```
Enter the previos deque:
```

```
8 5 7 1 4 6 3
```

```
^Z
```

```
Enter the image:
```

```
1
```

```
^Z
```

```
-----  
seque in a previos deque: image find 3
```

```
seque in a sorted deque: image find 0
```

```
1.00 3.00 4.00 5.00 6.00 7.00 8.00
```

5.2:

Данные, которые подаются в программу со стандартного ввода:

file.txt

[3.38 3.38 3.45 3.45] - образец

Содержимое файла:

file.txt
1 9.45 11.333 1.43 5.43 14.54 2.21 3.38 3.38 3.45 3.45 5.11 17.134

Представление данных основного дека в виде бинарного дерева:



Ожидаемый вывод:

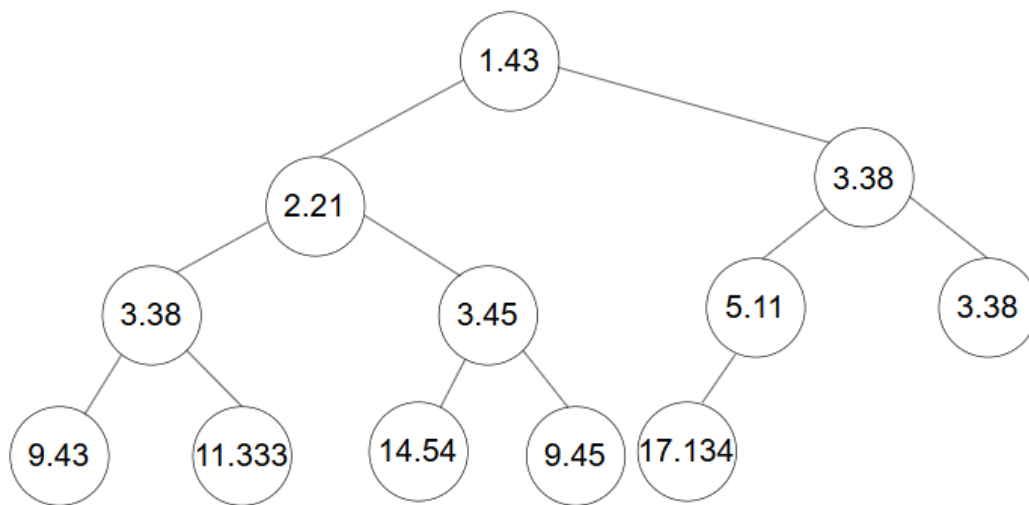
Расположение образца:

В исходном деке на 6 индексе.

В отсортированном на 2 индексе.

[1.43, 2.21, 3.38, 3.38, 3.45, 5.11, 5.43, 9.45, 11.333, 14.54, 17.134] - отсортированный дек.

Представление данных основного отсортированного дека в виде бинарного дерева:



Программный вывод:

```
write from StdI <= 0, FILE > 0.
```

```
2
```

```
file.txt
```

```
Your previous deque:
```

```
9.45 11.33 1.43 5.43 14.54 2.21 3.38 3.38 3.45 3.45 5.11 17.13
```

```
Enter the image:
```

```
3.38 3.38 3.45 3.45
```

```
^Z
```

```
-----  
seque in a previos deque: image find 6
```

```
seque in a sorted deque: image find 2
```

```
1.43 2.21 3.38 3.38 3.45 3.45 5.11 5.43 9.45 11.33 14.54 17.13
```


5.3

Данные, которые подаются в программу со стандартного ввода:

file.txt

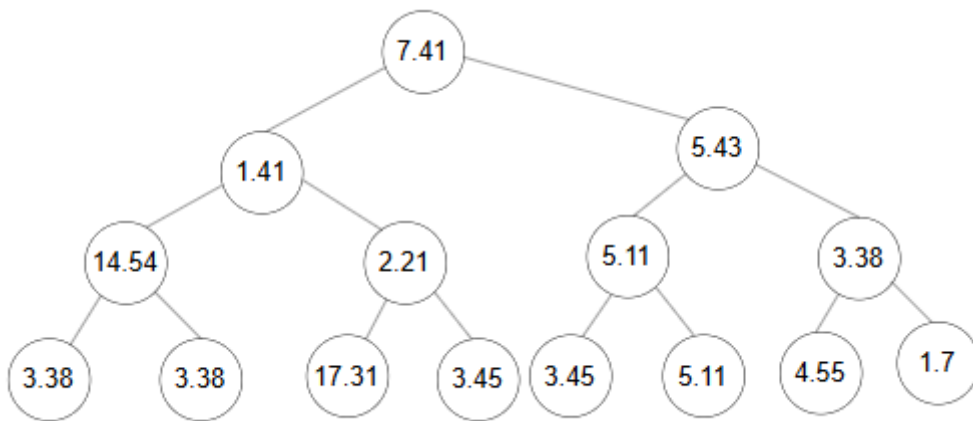
[12.46, 3.38] - образец

Содержимое файла:

file.txt

```
1 7.41 1.43 5.43 14.54 2.21 12.31 12.46 3.38 3.38 17.31 3.45 3.45 5.11 4.55 1.7
```

Представление данных основного дека в виде бинарного дерева:



Ожидаемый вывод:

Расположение образца:

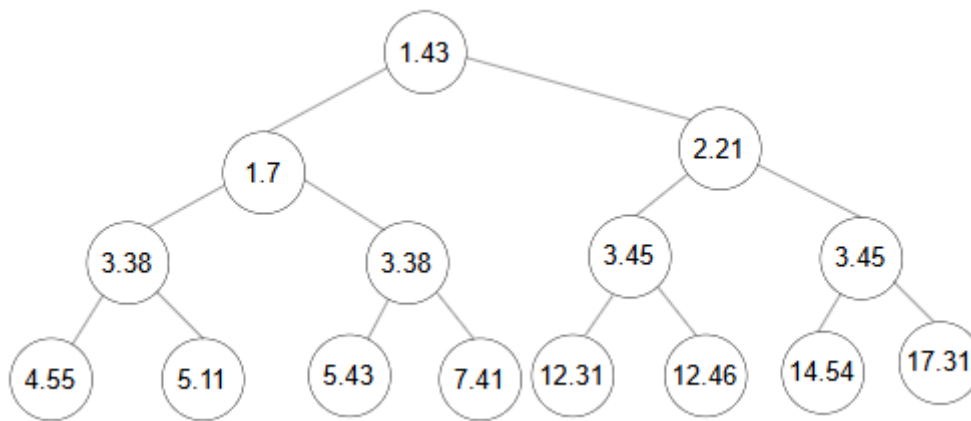
В исходном деке на 6 индексе.

В отсортированном отсутствует образ.

[1.43, 1.70, 2.21, 3.38, 3.38, 3.45, 3.45, 4.55, 5.11, 5.43, 7.41, 12.31, 12.46, 14.54, 17.31]

- отсортированный дек.

Представление данных основного отсортированного дека в виде бинарного дерева:



Программный вывод:

```
write from StdI <= 0, FILE > 0.
```

```
2
```

```
file.txt
```

```
Your previous deque:
```

```
7.41 1.43 5.43 14.54 2.21 12.31 12.46 3.38 3.38 17.31 3.45 3.45 5.11 4.55 1.70
```

```
Enter the image:
```

```
12.46 3.38
```

```
^Z
```

```
-----
```

```
seque in a previos deque: image find 6
```

```
seque in a sorted deque: image not find
```

```
1.43 1.70 2.21 3.38 3.38 3.45 3.45 4.55 5.11 5.43 7.41 12.31 12.46 14.54 17.31
```

6. Оценка сложности алгоритма.

6.1 Оценка сложности алгоритма для пирамидальной сортировки в данной программе.

Ключевые шаги алгоритма:

1. Построение максимальной кучи (функция `max_hear`).
2. Сортировка: извлечение максимального элемента и восстановление кучи (функция `pyramid_sort`).

Примечание: n — размер дека.

| 1. Построение максимальной кучи (функция `max_hear`)

Для построения максимальной кучи мы проходим по всем узлам, начиная с последнего узла, у которого есть дочерние узлы, и "просеиваем" его вниз.

Оценка сложности:

Чтобы оценить сложность функции `max_hear`, необходимо учитывать, что для произвольного доступа к элементам дека используется функция `getLn`. Эта функция выполняет проход по деку, и в худшем случае может потребоваться n проходов. В функции `max_hear` может быть выполнено максимум 4 таких прохода по n для каждого узла, что в итоге дает:

$$4 \cdot n \rightarrow O(n)$$

При этом в каждой куче примерно $n/2$ узлов являются листовыми, и для каждого из них мы выполняем операции "просеивания". Таким образом, общая сложность функции `max_hear` составляет:

$$O(n) \text{ (для одного узла)} \cdot O(n/2) \text{ (количество узлов)} = O(n^2)$$

| 2. Извлечение максимума (функция `pyramid_sort`)

Удаление максимального элемента из конца дека осуществляется за константное время $O(1)$ с помощью метода `pop_back`. Однако восстановление кучи требует произвольного доступа к элементам дека, который осуществляется с помощью функции `getLn`, уже оцененной нами как $O(n)$.

Затем снова вызывается функция `max_hear`, сложность которой составляет $O(n^2)$. Таким образом, суммарная сложность на каждом этапе извлечения максимума будет:

$$O(n) \text{ (доступ)} + O(n^2) \text{ (восстановление кучи)} = O(n^2 + n) = O(n^2)$$

Если этот процесс повторяется n раз (для каждого элемента), то общая сложность функции `pyramid_sort` составит:

$$O(n^2) \cdot n = O(n^3)$$

| Заключение

Таким образом, итоговая оценка временной сложности алгоритма пирамидальной сортировки с использованием дека и функции `getLn` составляет $O(n^3)$.

6.2 Оценка сложности алгоритма Кнута-Морриса-Пратта в данной программе.

Примечание:

- m — размер шаблона (образа) дека.
- n — размер основного дека.

| Ключевые шаги алгоритма:

1. Построение префикс-функции.
2. Поиск паттерна в основном деке.
1. Построение префикс-функции:

Начинаем с создания массива `prefix`, заполняя его нулями. Это требует $O(m)$ операций, так как мы выполняем m вставок.

Далее происходит заполнение префикс-функции:

Используем два указателя: один (i) для текущей позиции в шаблоне, другой (j) для отслеживания длины текущего совпадения.

В худшем случае для каждого i от 1 до m :

- Два вызова функции `getLn` требуют $O(i) + O(j)$, что в сумме составляет $O(m) + O(m)$ (так как i и j могут достигать значения m).
- Вставка в дек `prefix` с помощью `insert`, использующего функцию `getLn`, требует $O(i)$, что также в сумме составляет $O(m)$.
- Рекурсивные откаты j с использованием функции `getLn` суммарно требуют $O(m)$.

Итого: Общее время на построение префикс-функции составляет $O(m * m) = O(m^2)$.

2. Поиск паттерна в основном тексте:

Указатель i изменяется от 0 до n , что требует $O(n)$ операций.

Для каждого значения i происходит следующее:

- Вызов функции `getLn(seque, i)` требует $O(i)$, что в сумме составляет $O(n)$, так как i может достигать значения n .
- Вызов функции `getLn(img, j)` требует $O(j)$, где $j \leq m$, что в итоге составляет $O(m)$.
- Рекурсивные откаты j с использованием функции `getLn` суммарно требуют $O(m)$.

Итого: Общее время на поиск образа в деке составляет $O(n(n + m)) = O(n^2 + n \cdot m)$.

| Общая сложность алгоритма

Суммируя временные затраты на оба шага:

$$O(m^2) \text{ (построение префикс-функции)} + O(n^2 + n \cdot m) \text{ (поиск)} = O(m^2 + n^2 + n \cdot m)$$

Таким образом, окончательная оценка сложности алгоритма Кнута-Морриса-Пратта в данной программе составляет $O(m^2 + n^2 + n \cdot m)$.

Заключение

В ходе выполнения лабораторной работы было выполнено следующее техническое задание:

1. Разработана программа пирамидальной сортировки с последующим поиском методом Кнута-Морриса-Пратта, используя дек на базе связного списка. Также была оценена сложность обоих алгоритмов.

2. Создана программа на языке C++ для архитектуры x64 в среде Visual Studio Code 2022, реализующая указанные алгоритмы.

3. Проведены тесты для проверки корректности работы программы. Программа протестирована как в Windows, так и в Unix-подобных системах.

