

Министерство науки и высшего образования РФ  
Пензенский государственный университет  
Кафедра “Вычислительная техника”

## **Отчёт**

по лабораторной работе №3  
по курсу “Логика и основы алгоритмизации в инженерных задачах”  
на тему “Динамические списки”

Выполнил студент гр. 22ВВВ3:  
Кулахметов С.И.

Приняли:  
к.т.н., доцент Юрова О.В.  
к.э.н., доцент Акифьев И.В.

Пенза 2023

## Цель работы

Рассмотреть принципы работы динамических структур (односвязных списков), научиться реализовывать их на языке программирования Си. Выполнить лабораторное задание.

## Лабораторное задание

1. Реализовать приоритетную очередь, путём добавления элемента в список в соответствии с приоритетом объекта (т.е. объект с большим приоритетом становится перед объектом с меньшим приоритетом).

2. Реализовать структуру данных *Очередь*.

3. Реализовать структуру данных *Стек*.

## Пояснительный текст к программам

Лабораторное задание выполнено в виде трёх программ, каждая из которых отвечает за реализацию одной структуры данных.

В первой программе реализована структура данных *Приоритетная очередь*, которая, как и обычная очередь придерживается принципа FIFO, однако, данные извлекаются из неё в порядке наибольшего приоритета. Реализация данного алгоритма основана на односвязном списке, в основе которого лежит следующая структура данных:

```
struct Queue
{
    int data;    // Числовой элемент очереди
    int priority; // Приоритет элемента
    struct Queue *next; // Указатель на следующий элемент очереди
};
```

Во второй программе представлена реализация простой очереди, написанной по аналогии с первой программой, но без учёта приоритета при извлечении данных из структуры. Добавление происходит в конец списка, а чтение с удалением элемента — с начала.

Третья программа отвечает за реализацию структуры данных *Стек*, который придерживается принципа FILO. Добавление и чтение с удалением элемента происходят с конца списка.

## Результаты работы программ

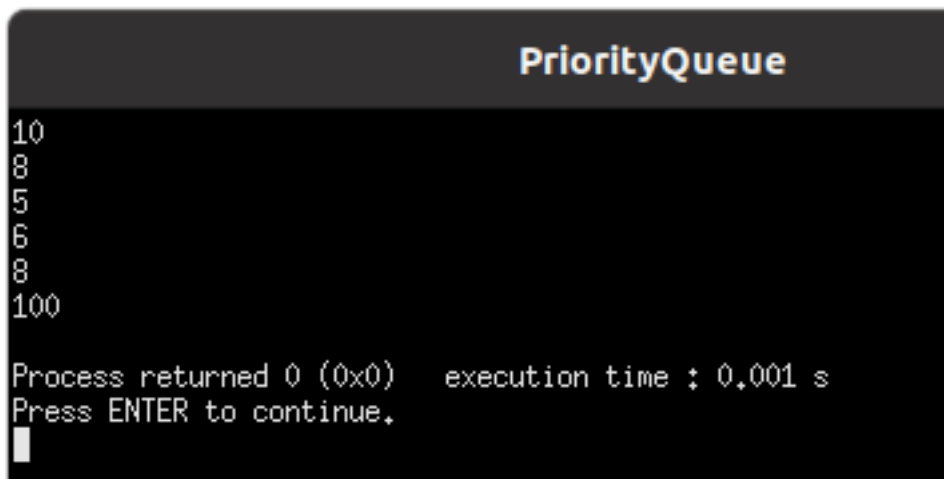
### Задание 1

Данные заносятся в приоритетную очередь в следующем порядке:

- 1) 10 (приоритет 11)
- 2) 5 (приоритет 3)
- 3) 8 (приоритет 1)
- 4) 6 (приоритет 3)
- 5) 8 (приоритет 6)
- 6) 100 (приоритет -2)

(см. Приложение Б рис. 1)

Результат извлечения данных из приоритетной очереди.



```
PriorityQueue
10
8
5
6
8
100
Process returned 0 (0x0)   execution time : 0.001 s
Press ENTER to continue.
█
```

Стоит обратить внимание на то, что если приоритет данных одинаковый, то они извлекаются в стандартном порядке обычной очереди.

### Задание 2

Данные заносятся в очередь в следующем порядке:

- 1) 10
- 2) 10
- 3) 5
- 4) 6
- 5) -70

(см. Приложение Б рис. 2)

Результат извлечения данных из очереди.

```
Lab3
10
10
5
6
-70
Process returned 0 (0x0)   execution time : 0.001 s
Press ENTER to continue.
```

### Задание 3

Данные заносятся в стек в следующем порядке:

- 1) 10
- 2) 5
- 3) 7
- 4) 8
- 5) -13
- 6) 1

(см. Приложение Б рис. 3)

Результат извлечения данных из стека.

```
Stack
1
-13
8
7
5
10
Process returned 0 (0x0)   execution time : 0.001 s
Press ENTER to continue.
```

### **Вывод**

В ходе выполнения данной лабораторной работы были получены навыки реализации динамических списков и основанных на них структур данных. На языке Си написаны: *Приоритетная очередь*, *Очередь* и *Стек*. Проверена их

работоспособность и учтён нюанс работы приоритетной очереди — данные с одинаковым приоритетом извлекаются в стандартном порядке очереди.

**Ссылка на *GitHub* репозиторий с лабораторной работой**

<https://github.com/KulakhmetovS/Lab3>

## Приложение А

### Листинг программ

#### Файл PriorityQueue/main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

struct Queue; // Структура, отвечающая за элементы очереди
struct Queue *init(int, int); // Инициализация очереди
void Push(struct Queue **, int, int); // Добавление элемента в очередь
int Priority(struct Queue *); // Сортировка элементов в соответствии с их приоритетом
struct Queue *Pop(struct Queue *); // Чтение элемента из очереди с последующим удалением

int res = 0; // Результат извлечения из очереди
int *array; // Указатель на массив с приоритетами

int main()
{
    int a = 10;

    // Третьим параметром Push() введите приоритет элемента
    struct Queue *list = init(a, 11); // Инициализация очереди
    Push(&list, 5, 3); // Добавление значения в очередь
    Push(&list, 8, 1);
    Push(&list, 6, 3);
    Push(&list, 8, 6);
    Push(&list, 100, -2);
    list = Pop(list); // Получение значения и нового указателя
    printf("%d\n", res);
    list = Pop(list);
    printf("%d\n", res);
    list = Pop(list);
    printf("%d\n", res);
    list = Pop(list);
    printf("%d\n", res);
    list = Pop(list);
    printf("%d\n", res);
    list = Pop(list);
    printf("%d\n", res);

    return 0;
}

struct Queue
{
    int data; // Числовой элемент очереди
    int priority; // Приоритет элемента
    struct Queue *next; // Указатель на следующий элемент очереди
```

```

};

struct Queue *init(int element, int prt)
{
    struct Queue *p = NULL; // Создание указателя на структуру

    if((p = malloc(sizeof(struct Queue))) == NULL) // Выделение
памяти под структуру
    {
        printf("Unable to allocate memory: ");
        exit(1);
    }

    p -> data = element;      // Присваивание введённого значения
полю данных
    p -> priority = prt;
    p -> next = NULL;      // Установка на нулевой указатель

    return p;
}

void Push(struct Queue **list ,int element, int prt)
{
    struct Queue *tmp = *list; // Сохранение оригинального
указателя на голову

    if(tmp != NULL) // Проверка на существование списка
    {
        struct Queue *new_element = init(element, prt); // Создание
нового элемента

        while(tmp -> next != NULL)
        {
            tmp = tmp -> next;
        }

        tmp -> next = new_element;
    }
    else if(tmp == NULL) // Инициализация списка, если его нет
    {
        *list = init(element, prt);
    }
}

int Priority(struct Queue *list)
{
    int n = 0, j = 0;
    int num = 1;
    struct Queue *tmp = list;

    // Узнаём количество элементов в списке
    while(tmp -> next != NULL)
    {
        num++;
        tmp = tmp -> next;
    }
}

```

```

    tmp = list;

    array = calloc(num, 4); // Создаём массив

    for(j = 0; j < num; j++)
    {
        array[j] = tmp -> priority; // Запись в массив
приоритетов
        tmp = tmp -> next;
    }
    tmp = list;

    // Сортировка массива пузырьком (от большего к меньшему)

    if(num > 1)
    {
        for(j = 0; j < num - 1; j++)
        {
            for(int k = 0; k < num - 1; k++)
            {
                if(array[k] < array[k + 1])
                {
                    n = array[k];
                    array[k] = array[k + 1];
                    array[k + 1] = n;
                }
            }
        }

        /*for(j = 0; j < num; j++)
        printf("%d ", array[j]);*/

        return num;
    }

    struct Queue *Pop(struct Queue *list)
    {
        int i = Priority(list);
        struct Queue *head = list, *prev;

        if(head -> priority == array[0]) // Если элемент первый
        {
            res = head -> data;
            list = head -> next; //Переназначение первого
указателя на следующий
            free(head);
            head = list;
            return list;
        }
        else
        {
            prev = list; // Указатель на предыдущий элемент
            head = list -> next; // Указатель на следующий элемент
        }
    }

```



```

for(int j = 0; j < i; j++) // Если элемент в середине
{
    if(head -> priority == array[0])
    {
        res = head -> data;
        if(head -> next != NULL)
        {
            prev -> next = head -> next;
            free(head);
            head = prev -> next;
            return list;
        }
        else // Если элемент в конце
        {
            prev -> next = NULL;
            free(head);
            return list;
        }
    }
    else
    {
        prev = head -> next;
        head = prev -> next;
    }
}

return list;
}

```

#### **Файл Queue/main.c**

```

#include <stdio.h>
#include <stdlib.h>

```

```

struct Queue; // Структура, отвечающая за элементы очереди
struct Queue *init(int); // Инициализация очереди
void Push(struct Queue **, int); // Добавление элемента в очередь
struct Queue *Pop(struct Queue *); // Чтение элемента из очереди с последующим удалением

```

```

int res = 0; // Результат извлечения из очереди

```

```

int main()
{

```

```

    int a = 10;

```

```

    struct Queue *list = init(a); // Инициализация очереди

```

```

    Push(&list, a); // Добавление значения в очередь

```

```

    Push(&list, 5);

```

```

    Push(&list, 6);

```

```

    Push(&list, -70);

```

```

    list = Pop(list); // Получение значения и нового указателя

```

очереди

```

    printf("%d\n", res);

```

```

    list = Pop(list);

```

```

    printf("%d\n", res);

```

```

        list = Pop(list);
        printf("%d\n", res);
        list = Pop(list);
        printf("%d\n", res);
        list = Pop(list);
        printf("%d\n", res);

    return 0;
}

struct Queue
{
    int data;    // Числовой элемент очереди
    struct Queue *next; // Указатель на следующий элемент очереди
};

struct Queue *init(int element)
{
    struct Queue *p = NULL; // Создание указателя на структуру

    if((p = malloc(sizeof(struct Queue))) == NULL) // Выделение
памяти под структуру
    {
        printf("Unable to allocate memory: ");
        exit(1);
    }

    p -> data = element;    // Присваивание введённого значения
полю данных
    p -> next = NULL;    // Установка на нулевой указатель

    return p;
}

void Push(struct Queue **list ,int element)
{
    struct Queue *tmp = *list;    // Сохранение оригинального
указателя на голову

    if(tmp != NULL) // Проверка на существование списка
    {
        struct Queue *new_element = init(element);    // Создание
нового элемента

        while(tmp -> next != NULL)
        {
            tmp = tmp -> next;
        }

        tmp -> next = new_element;
    }
    else if(tmp == NULL)    // Инициализация списка, если его нет
    {
        *list = init(element);
    }
}

```

```

    }
}

struct Queue *Pop(struct Queue *list)
{
    res = list -> data; // Получение элемента очереди

    struct Queue *to_delete = list; // Переназначение указателя
на первый элемент
    list = list -> next;    //Переназначение первого указателя на
следующий
    free(to_delete);        // Очистка памяти по предыдущему
указателю

    return list;           // Возвращение нового указателя на вершину
очереди
}

```

### **Файл Stack/main.c**

```

#include <stdio.h>
#include <stdlib.h>

struct Stack; // Структура, отвечающая за элементы очереди
struct Stack *init(int); // Инициализация очереди
void Push(struct Stack **, int); // Добавление элемента в
очередь
int Pop(struct Stack *); // Чтение элемента из очереди с
последующим удалением

int main()
{
    int a = 10;
    int result = 0;

    struct Stack *list = init(a); // Инициализация очереди
    Push(&list, 5); // Добавление значения в очередь
    Push(&list, 7);
    Push(&list, 8);
    Push(&list, -13);
    Push(&list, 1);
    result = Pop(list); // Получение значения
    printf("%d\n", result);
    result = Pop(list);
    printf("%d\n", result);
    result = Pop(list);
    printf("%d\n", result);
    result = Pop(list);
    printf("%d\n", result);
    result = Pop(list);
    printf("%d\n", result);
    result = Pop(list);
    printf("%d\n", result);
}

```

```

        return 0;
    }

    struct Stack
    {
        int data;    // Числовой элемент очереди
        struct Stack *next; // Указатель на следующий элемент очереди
    };

    struct Stack *init(int element)
    {
        struct Stack *p = NULL; // Создание указателя на структуру

        if((p = malloc(sizeof(struct Stack))) == NULL) // Выделение
памяти под структуру
        {
            printf("Unable to allocate memory: ");
            exit(1);
        }

        p -> data = element;    // Присваивание введённого значения
полю данных
        p -> next = NULL;    // Установка на нулевой указатель

        return p;
    }

    void Push(struct Stack **list ,int element)
    {
        struct Stack *new_element = init(element);    // Создание
нового элемента

        struct Stack *tmp = *list;    // Сохранение оригинального
указателя на голову

        while(tmp -> next != NULL)
        {
            tmp = tmp -> next;
        }

        tmp -> next = new_element;
    }

    int Pop(struct Stack *list)
    {
        int res = 0;    // Результат извлечения из очереди
        int i = 0, j;    // Переменные счётчики числа элементов стека
        struct Stack *tmp = list, *p = list;    // Сохранение
указателя на вершину стека

        while(list -> next != NULL) // Обход списка
        {
            list = list -> next;
            i++;
        }
    }

```

```

    i--;

    res = list -> data; // Получение элемента стека

    struct Stack *to_delete = list; // Переназначение указателя
на последний элемент

    for(j = 0; j < i; j++)
    {
        tmp = tmp -> next;          // Очередной обход списка до
предпоследнего элемента
    }

    tmp -> next = NULL; // Присвоение предпоследнему элементу
нулевого указателя

    // Сохранение первого листа списка
    if(p -> next != NULL) free(to_delete);    // Очистка памяти
по последнему указателю

    return res;    // Возвращение значения
}

```

## Приложение Б

### Внесение данных в структуры

```
struct Queue *list = init(a, 11); // Инициализация очереди
Push(&list, 5, 3); // Добавление значения в очередь
Push(&list, 8, 1);
Push(&list, 6, 3);
Push(&list, 8, 6);
Push(&list, 100, -2);
```

Рисунок 1 — Внесение данных в приоритетную очередь

```
struct Queue *list = init(a); // Инициализация очереди
Push(&list, a); // Добавление значения в очередь
Push(&list, 5);
Push(&list, 6);
Push(&list, -70);
```

Рисунок 2 — Внесение данных в очередь

```
struct Stack *list = init(a); // Инициализация очереди
Push(&list, 5); // Добавление значения в очередь
Push(&list, 7);
Push(&list, 8);
Push(&list, -13);
Push(&list, 1);
```

Рисунок 3 — Внесение данных в стек