

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.

Так как все алгоритмы реализованы через динамические списки, то после ввода элемента выводится запрос на повторение внесения данных в структуру.

Результаты работы программ

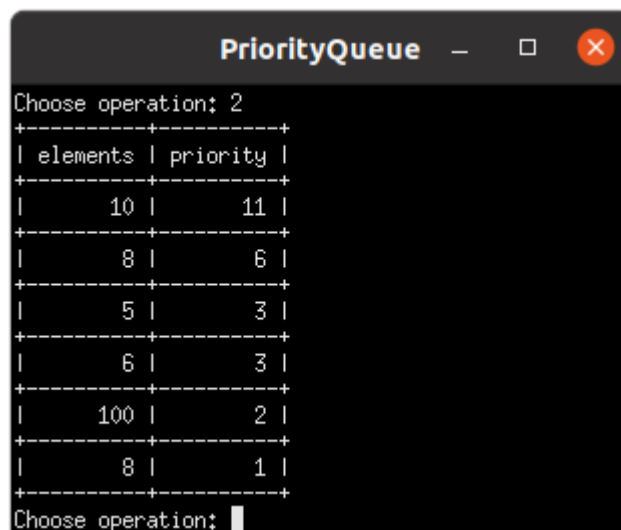
Задание 1

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.



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

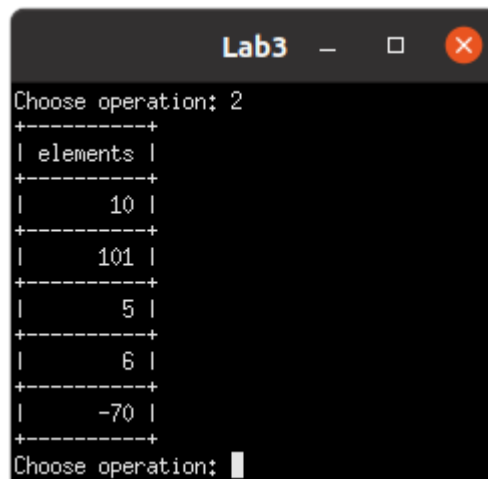
Задание 2

Данные заносятся в очередь в следующем порядке:

- 1) 10
- 2) 101
- 3) 5
- 4) 6
- 5) -70

(см. Приложение Б рис. 2)

Результат извлечения данных из очереди.



```
Lab3
Choose operation: 2
+-----+
| elements |
+-----+
|    10    |
+-----+
|   101    |
+-----+
|     5    |
+-----+
|     6    |
+-----+
|   -70    |
+-----+
Choose operation: 
```

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
Choose operation: 2
+-----+
| elements |
+-----+
|     1    |
+-----+
|   -13    |
+-----+
|     8    |
+-----+
|     7    |
+-----+
|     5    |
+-----+
|    10    |
+-----+
Choose operation: 
```

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>

struct Queue; // Структура, отвечающая за элементы очереди
struct Queue *init(int, int); // Инициализация очереди
void Push(struct Queue **, int, int); // Добавление элемента в очередь
struct Queue *Priority(struct Queue *); // Сортировка элементов в соответствии с их приоритетом
struct Queue *QueueSort(struct Queue *, int); // Создание отсортированной очереди
struct Queue *Pop(struct Queue *); // Чтение элемента из очереди
struct Queue *DeleteElement(struct Queue *, int, int); // Удаление элемента
void DeleteQueue(struct Queue *); // Удаление всей очереди

int res = 0; // Результат извлечения из очереди
int pr = 0; // Результат извлечения приоритета
int array[1000]; // Указатель на массив с приоритетами

int main()
{
    float operation = 0;
    struct Queue *list = NULL, *sortq = NULL;
    int element = 0, priority = 0;

    printf(" # Priority Queue #\n\n\tMenu\n");
    printf(" 1 - add new element\n 2 - see list\n 3 - delete element\n 4 - delete queue\n 0 - quit\n");

    while(1)
    {
        label:
        printf("Choose operation: ");
        scanf("%f", &operation);
        if((operation > 4) || (operation < 0))
        {
            printf("Invalid operation! Try again\n");
            goto label;
        }
        else if(operation == 0) // Окончание программы
        {
            break;
        }
        else if(operation == 1) // Ввод элементов в приоритетную очередь
        {
            entering:
            printf("Enter element(any integer) and priority(>0): ");
            scanf("%d%d", &element, &priority);
            if(priority <= 0)
            {
                printf("priority must be more than 0!\n");
                goto entering;
            }
            Push(&list, element, priority);
        }
        else if(operation == 2) // Вывод всей очереди
        {
            if(list == NULL)
            {
                printf("list is empty\n");
                goto label;
            }
        }
    }
}
```

```

        }
        list = Priority(list);
        sortq = list;
        printf("+-----+-----+\n| elements | priority |\n+-----+-----+\n");
        while(list != NULL)
        {
            list = Pop(list);
            printf("|%9d |%9d |\n+-----+-----+\n", res, pr);
        }
        list = sortq;
    }
    else if(operation == 3) // Удвление элемента очереди
    {
        printf("Enter element and priority you need to delete: ");
        scanf("%d%d", &element, &priority);
        list = DeleteElement(list, element, priority);
    }
    else if(operation == 4) // Удвление всей очереди
    {
        list = Priority(list);
        DeleteQueue(list);
        list = NULL;
    }
    else
    {
        printf("Invalid operation\n");
        goto label;
    }
}

return 0;
}

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

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

    p = malloc(sizeof(struct Queue)); // Выделение памяти под структуру

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

    return p;
}

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

    if(tmp != NULL) // Проверка на существование списка
    {
        struct Queue *new_element = init(element, prio); // 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, prio);
}
}

```

```

struct Queue *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 приоритетов

```

```

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

```

```

    list = QueueSort(list, num);    // Получение указателя на отсортированный

```

список

```

    return list;
}

```

```

struct Queue *QueueSort(struct Queue *list, int size)
{

```

```

    struct Queue *tmp = list, *pointer = NULL;
    int i = 0;    // Создаём массив для элементов

```

```

    // Проходим по массиву и списку, сравнивая приоритеты
    for(i = 0; i < size; i++)
    {
        tmp = list;
        while(tmp != NULL)
        {

```



```

        if(tmp -> priority == array[i])
        {
            Push(&pointer, tmp -> data, array[i]); // Запись элементов и
приоритетов в новую очередь
            tmp -> priority = 0; // Обнуление прочитанного приоритета
        }
        tmp = tmp -> next;
    }
}

// Освобождаем память
//free(arr);
//free(array);

while(list != NULL)
{
    struct Queue *to_delete = list;
    list = list -> next;
    free(to_delete);
}

return pointer;
}

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

    //struct Queue *to_delete = sortq; // 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
}

struct Queue *DeleteElement(struct Queue *list, int element, int prio)
{
    if(list == NULL)
    {
        printf("List is empty");
        return list;
    }

    struct Queue *tmp = list, *head = NULL, *prev = NULL;
    int flag = 0;

    if((tmp -> data == element) && (tmp -> priority == prio))
    {
        head = tmp;
        tmp = tmp -> next;
        free(head);
        list = tmp;
        flag = 1;
    }
    else
    {
        prev = tmp;
        head = tmp -> next;
    }

    while(head != NULL)
    {
        if((head -> data == element) && (head -> priority == prio))
        {
            if(head -> next != NULL)
            {

```

```

        prev -> next = head -> next;
        free(head);
        head = prev -> next;
        flag = 1;
    }
    else
    {
        prev -> next = NULL;
        free(head);
        flag = 1;
    }
}
else
{
    prev = head;
    head = head -> next;
}
}

if(flag == 0)
{
    printf("Element not found!\n");
}

return list;
}

void DeleteQueue(struct Queue *list)
{
    struct Queue *to_delete = NULL;
    while(list != NULL)
    {
        to_delete = list; // Переназначение указателя на первый элемент
        list = list -> next; //Переназначение первого указателя на
следующий
        free(to_delete); // Очистка памяти по предыдущему указателю
    }
    printf("list deleted\n");
}

```

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
struct Queue *DeleteElement(struct Queue *, int); // Удаление элемента
void DeleteQueue(struct Queue *); // Удаление всей очереди

int res = 0; // Результат извлечения из очереди

int main()
{
    int elem = 0;
    float operation = 0;
    struct Queue *list = NULL, *queue = NULL;

    printf(" # Queue #\n\n\tMenu\n");
    printf(" 1 - add new element\n 2 - see list\n 3 - delete element\n 4 -
delete queue\n 0 - quit\n");
}

```

```

while(1)
{
    label:
    printf("Choose operation: ");
    scanf("%f", &operation);
    if((operation > 4) || (operation < 0))
    {
        printf("Invalid operation! Try again\n");
        goto label;
    }
    else if(operation == 0) // Окончание программы
    {
        break;
    }
    else if(operation == 1) // Ввод элементов в приоритетную очередь
    {
        printf("Enter element(any integer): ");
        scanf("%d", &elem);
        Push(&list, elem);
    }
    else if(operation == 2) // Вывод всей очереди
    {
        if(list == NULL)
        {
            printf("list is empty\n");
            goto label;
        }
        queue = list;
        printf("+-----+\n| elements |\n+-----+\n");
        while(list != NULL)
        {
            list = Pop(list);
            printf("|%9d |\n+-----+\n", res);
        }
        list = queue;
    }
    else if(operation == 3) // Удвление элемента очереди
    {
        printf("Enter element you need to delete: ");
        scanf("%d", &elem);
        list = DeleteElement(list, elem);
    }
    else if(operation == 4) // Удвление всей очереди
    {
        DeleteQueue(list);
        list = NULL;
    }
    else
    {
        printf("Invalid operation\n");
        goto label;
    }
}

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

    list = list -> next;    //Переназначение первого указателя на следующий

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

struct Queue *DeleteElement(struct Queue *list, int elem)
{
    if(list == NULL)
    {
        printf("List is empty");
        return list;
    }

    struct Queue *tmp = list, *head = NULL, *prev = NULL;
    int flag = 0;

    if(tmp -> data == elem)
    {
        head = tmp;
        tmp = tmp -> next;
        free(head);
        list = tmp;
        flag = 1;
    }
    else
    {
        prev = tmp;
        head = tmp -> next;
    }
}

```

```

    }

    while(head != NULL)
    {
        if(head -> data == elem)
        {
            if(head -> next != NULL)
            {
                prev -> next = head -> next;
                free(head);
                head = prev -> next;
                flag = 1;
            }
            else
            {
                prev -> next = NULL;
                free(head);
                flag = 1;
            }
        }
        else
        {
            prev = head;
            head = head -> next;
        }
    }

    if(flag == 0)
    {
        printf("Element not found!\n");
    }

    return list;
}

void DeleteQueue(struct Queue *list)
{
    struct Queue *to_delete = NULL;
    while(list != NULL)
    {
        to_delete = list; // Переназначение указателя на первый элемент
        list = list -> next; //Переназначение первого указателя на
следующий
        free(to_delete); // Очистка памяти по предыдущему указателю
    }
    printf("list deleted\n");
}

```

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
void Printing(struct Stack *);
struct Stack *DeleteElement(struct Stack *, int); // Удаление элемента
void DeleteStack(struct Stack *); // Удаление всей очереди

int main()
{

```

```

int element = 0;
struct Stack *list = NULL, *pointer = NULL;
float operation = 0;

printf(" # Stack #\n\n\tMenu\n");
printf(" 1 - add new element\n 2 - see list\n 3 - delete element\n 4 -
delete queue\n 0 - quit\n");

while(1)
{
    label:
    printf("Choose operation: ");
    scanf("%f", &operation);
    if((operation > 4) || (operation < 0))
    {
        printf("Invalid operation! Try again\n");
        goto label;
    }
    else if(operation == 0) // Окончание программы
    {
        break;
    }
    else if(operation == 1) // Ввод элементов в приоритетную очередь
    {
        printf("Enter element(any integer): ");
        scanf("%d", &element);
        Push(&list, element);
    }
    else if(operation == 2) // Вывод всей очереди
    {
        if(list == NULL)
        {
            printf("list is empty\n");
            goto label;
        }
        pointer = list;
        Printing(list);
        list = pointer;
    }
    else if(operation == 3) // Удвление элемента очереди
    {
        printf("Enter element you need to delete: ");
        scanf("%d", &element);
        list = DeleteElement(list, element);
    }
    else if(operation == 4) // Удвление всей очереди
    {
        DeleteStack(list);
        list = NULL;
    }
    else
    {
        printf("Invalid operation\n");
        goto label;
    }
}

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

    if(tmp != NULL)
    {
        struct Stack *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);
    }
}

void Printing(struct Stack *list)
{
    int i = 0, j; // Переменные счётчики числа элементов стека
    struct Stack *tmp = list; // Сохранение указателя на вершину стека

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

    tmp = list;
    printf("+-----+\n| elements |\n+-----+\n");
    while(i >= 0)
    {
        for(j = 0; j < i; j++)
        {
            tmp = tmp -> next;
        }
        printf("|%9d |\n+-----+\n", tmp -> data);
        tmp = list;
        i--;
    }
}

void DeleteStack(struct Stack *list)
{

```

```

    struct Stack *to_delete = NULL;
    while(list != NULL)
    {
        to_delete = list; // Переназначение указателя на первый элемент
        list = list -> next; //Переназначение первого указателя на
следующий
        free(to_delete); // Очистка памяти по предыдущему указателю
    }
    printf("list deleted\n");
}

struct Stack *DeleteElement(struct Stack *list, int elem)
{
    if(list == NULL)
    {
        printf("List is empty");
        return list;
    }

    struct Stack *tmp = list, *head = NULL, *prev = NULL;
    int flag = 0;

    if(tmp -> data == elem)
    {
        head = tmp;
        tmp = tmp -> next;
        free(head);
        list = tmp;
        flag = 1;
    }
    else
    {
        prev = tmp;
        head = tmp -> next;
    }

    while(head != NULL)
    {
        if(head -> data == elem)
        {
            if(head -> next != NULL)
            {
                prev -> next = head -> next;
                free(head);
                head = prev -> next;
                flag = 1;
            }
            else
            {
                prev -> next = NULL;
                free(head);
                flag = 1;
            }
        }
        else
        {
            prev = head;
            head = head -> next;
        }
    }

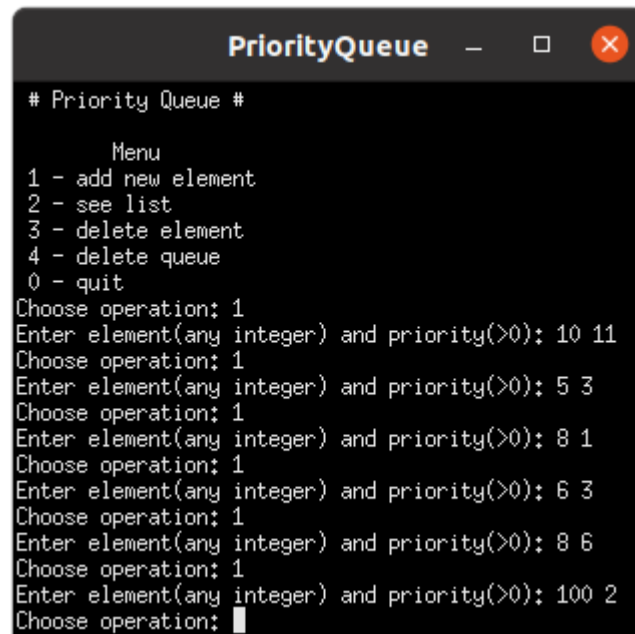
    if(flag == 0)
    {
        printf("Element not found!\n");
    }

    return list;
}

```


Приложение Б

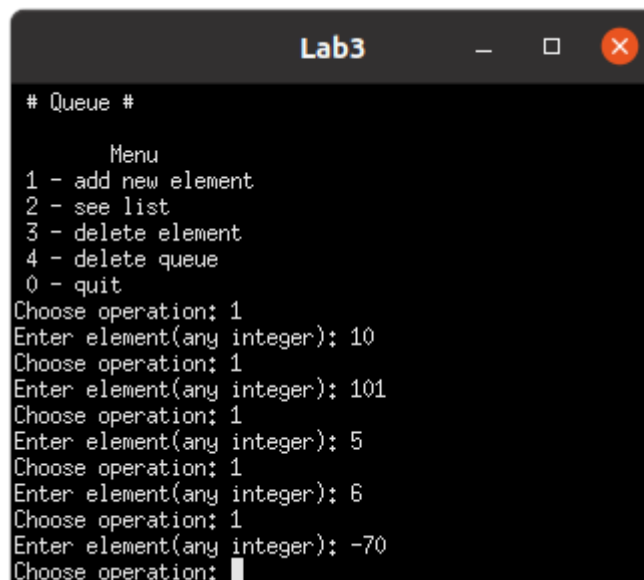
Внесение данных в структуры



```
PriorityQueue
# Priority Queue #

Menu
1 - add new element
2 - see list
3 - delete element
4 - delete queue
0 - quit
Choose operation: 1
Enter element(any integer) and priority(>0): 10 11
Choose operation: 1
Enter element(any integer) and priority(>0): 5 3
Choose operation: 1
Enter element(any integer) and priority(>0): 8 1
Choose operation: 1
Enter element(any integer) and priority(>0): 6 3
Choose operation: 1
Enter element(any integer) and priority(>0): 8 6
Choose operation: 1
Enter element(any integer) and priority(>0): 100 2
Choose operation: █
```

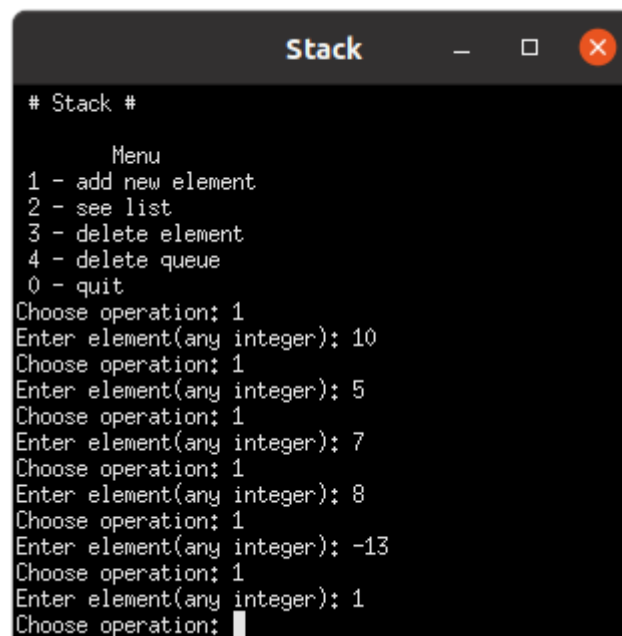
Рисунок 1 — Внесение данных в приоритетную очередь



```
Lab3
# Queue #

Menu
1 - add new element
2 - see list
3 - delete element
4 - delete queue
0 - quit
Choose operation: 1
Enter element(any integer): 10
Choose operation: 1
Enter element(any integer): 101
Choose operation: 1
Enter element(any integer): 5
Choose operation: 1
Enter element(any integer): 6
Choose operation: 1
Enter element(any integer): -70
Choose operation: █
```

Рисунок 2 — Внесение данных в очередь



```
Stack
# Stack #
      Menu
1 - add new element
2 - see list
3 - delete element
4 - delete queue
0 - quit
Choose operation: 1
Enter element(any integer): 10
Choose operation: 1
Enter element(any integer): 5
Choose operation: 1
Enter element(any integer): 7
Choose operation: 1
Enter element(any integer): 8
Choose operation: 1
Enter element(any integer): -13
Choose operation: 1
Enter element(any integer): 1
Choose operation: █
```

Рисунок 3 — Внесение данных в стек