**Министерство науки и высшего образования РФ**

**Пензенский государственный университет**

**Кафедра «Вычислительная техника»**

**ОТЧЁТ**

По лабораторной работе №3

По дисциплине «Логика и основы алгоритмизации в инженерных задачах»

На тему «Динамические списки»

**Выполнил студент группы 24ВВВ2:**

Лисицын Н.С.

**Приняли:**

Юрова О.В.

Деев М.В.

**Пенза, 2025**

**Цель работы**

Научиться реализовывать алгоритмы включающие в себя динамические списки.

**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. На основе приведенного в задании кода реализуйте структуру данных Очередь.
3. На основе приведенного в задании кода реализовать структуру данных Стек.

**Описание метода решения задачи**

Для обращения к динамическим данным применим указатели. Набор операций над списком будет включать добавление и удаление элементов, поиск элементов списка.

Используем динамическую реализацию списка. Для создания элемента списка используется структура, в которой объединяются полезная информация и ссылка на следующий элемент списка.

В первом задании переменная priority будет хранить приоритет элемента, и в зависимости от приоритета элемент будет помещен на место в списке.

Для реализации очереди используем принцип first in first out, для реализации стека используем принцип last in first out

**Листинг программы**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <locale.h>

struct node {

char inf[256];

int priority;

struct node\* next;

};

struct node\* priority\_head = NULL;

struct node\* queue\_front = NULL;

struct node\* queue\_rear = NULL;

struct node\* stack\_top = NULL;

struct node\* get\_priority\_struct(void);

void insert\_by\_priority(void);

void display\_priority(void);

void delete\_highest\_priority(void);

void destroy\_priority\_queue(void);

struct node\* get\_queue\_struct(void);

void enqueue(void);

void dequeue(void);

void display\_queue(void);

void destroy\_queue(void);

struct node\* get\_stack\_struct(void);

void push(void);

void pop(void);

void display\_stack(void);

void destroy\_stack(void);

void display\_menu(void);

int main() {

setlocale(LC\_ALL, "");

int choice, structure\_choice;

while (1) {

display\_menu();

scanf\_s("%d", &structure\_choice);

switch (structure\_choice) {

case 1: // Приоритетная очередь

while (1) {

printf("\n=== ПРИОРИТЕТНАЯ ОЧЕРЕДЬ ===\n");

printf("1. Добавить элемент с приоритетом\n");

printf("2. Просмотреть очередь\n");

printf("3. Удалить элемент с наивысшим приоритетом\n");

printf("4. Очистить очередь\n");

printf("5. Вернуться в главное меню\n");

printf("Выберите действие: ");

scanf\_s("%d", &choice);

if (choice == 5) break;

switch (choice) {

case 1: insert\_by\_priority(); break;

case 2: display\_priority(); break;

case 3: delete\_highest\_priority(); break;

case 4: destroy\_priority\_queue(); break;

default: printf("Неверный выбор\n");

}

}

break;

case 2: // Обычная очередь

while (1) {

printf("\n=== ОБЫЧНАЯ ОЧЕРЕДЬ (FIFO) ===\n");

printf("1. Добавить элемент в конец (enqueue)\n");

printf("2. Удалить элемент из начала (dequeue)\n");

printf("3. Просмотреть очередь\n");

printf("4. Очистить очередь\n");

printf("5. Вернуться в главное меню\n");

printf("Выберите действие: ");

scanf\_s("%d", &choice);

if (choice == 5) break;

switch (choice) {

case 1: enqueue(); break;

case 2: dequeue(); break;

case 3: display\_queue(); break;

case 4: destroy\_queue(); break;

default: printf("Неверный выбор\n");

}

}

break;

case 3: // Стек

while (1) {

printf("\n=== СТЕК (LIFO) ===\n");

printf("1. Добавить элемент в вершину (push)\n");

printf("2. Удалить элемент из вершины (pop)\n");

printf("3. Просмотреть стек\n");

printf("4. Очистить стек\n");

printf("5. Вернуться в главное меню\n");

printf("Выберите действие: ");

scanf\_s("%d", &choice);

if (choice == 5) break;

switch (choice) {

case 1: push(); break;

case 2: pop(); break;

case 3: display\_stack(); break;

case 4: destroy\_stack(); break;

default: printf("Неверный выбор\n");

}

}

break;

case 4:

// Очистка всех структур перед выходом

destroy\_priority\_queue();

destroy\_queue();

destroy\_stack();

printf("Выход из программы.\n");

exit(0);

default:

printf("Неверный выбор\n");

}

}

return 0;

}

void display\_menu(void) {

printf("\n=== ГЛАВНОЕ МЕНЮ ===\n");

printf("1. Работа с приоритетной очередью\n");

printf("2. Работа с обычной очередью (FIFO)\n");

printf("3. Работа со стеком (LIFO)\n");

printf("4. Выход\n");

printf("Выберите структуру данных: ");

}

// ==================== ПРИОРИТЕТНАЯ ОЧЕРЕДЬ ====================

struct node\* get\_priority\_struct(void) {

struct node\* p = (struct node\*)malloc(sizeof(struct node));

char s[256];

int prio;

if (p == NULL) {

printf("Ошибка при распределении памяти\n");

exit(1);

}

printf("Введите название объекта: ");

scanf\_s("%s", s, (unsigned)sizeof(s));

if (\*s == 0) {

printf("Запись не была произведена\n");

free(p);

return NULL;

}

printf("Введите приоритет (целое число, чем больше - тем выше приоритет): ");

scanf\_s("%d", &prio);

strcpy\_s(p->inf, sizeof(p->inf), s);

p->priority = prio;

p->next = NULL;

return p;

}

void insert\_by\_priority(void) {

struct node\* p = get\_priority\_struct();

if (p == NULL) return;

struct node\* current = priority\_head;

struct node\* prev = NULL;

if (priority\_head == NULL || p->priority > priority\_head->priority) {

p->next = priority\_head;

priority\_head = p;

printf("Элемент '%s' с приоритетом %d добавлен\n", p->inf, p->priority);

return;

}

while (current != NULL && current->priority >= p->priority) {

prev = current;

current = current->next;

}

prev->next = p;

p->next = current;

printf("Элемент '%s' с приоритетом %d добавлен\n", p->inf, p->priority);

}

void display\_priority(void) {

struct node\* current = priority\_head;

if (priority\_head == NULL) {

printf("Приоритетная очередь пуста\n");

return;

}

printf("\n=== ПРИОРИТЕТНАЯ ОЧЕРЕДЬ ===\n");

printf("Приоритет | Данные\n");

printf("----------|--------\n");

while (current != NULL) {

printf("%9d | %s\n", current->priority, current->inf);

current = current->next;

}

}

void delete\_highest\_priority(void) {

if (priority\_head == NULL) {

printf("Приоритетная очередь пуста\n");

return;

}

struct node\* temp = priority\_head;

printf("Удален элемент: '%s' с приоритетом %d\n", temp->inf, temp->priority);

priority\_head = priority\_head->next;

free(temp);

}

void destroy\_priority\_queue(void) {

struct node\* current = priority\_head;

struct node\* next;

int count = 0;

while (current != NULL) {

next = current->next;

free(current);

current = next;

count++;

}

priority\_head = NULL;

printf("Приоритетная очередь очищена. Удалено элементов: %d\n", count);

}

// ==================== ОБЫЧНАЯ ОЧЕРЕДЬ (FIFO) ====================

struct node\* get\_queue\_struct(void) {

struct node\* p = (struct node\*)malloc(sizeof(struct node));

char s[256];

if (p == NULL) {

printf("Ошибка при распределении памяти\n");

exit(1);

}

printf("Введите название объекта: ");

scanf\_s("%s", s, (unsigned)sizeof(s));

if (\*s == 0) {

printf("Запись не была произведена\n");

free(p);

return NULL;

}

strcpy\_s(p->inf, sizeof(p->inf), s);

p->next = NULL;

return p;

}

void enqueue(void) {

struct node\* p = get\_queue\_struct();

if (p == NULL) return;

if (queue\_rear == NULL) {

queue\_front = queue\_rear = p;

}

else {

queue\_rear->next = p;

queue\_rear = p;

}

printf("Элемент '%s' добавлен в конец очереди\n", p->inf);

}

void dequeue(void) {

if (queue\_front == NULL) {

printf("Очередь пуста\n");

return;

}

struct node\* temp = queue\_front;

printf("Удален элемент из начала очереди: '%s'\n", temp->inf);

queue\_front = queue\_front->next;

if (queue\_front == NULL) {

queue\_rear = NULL;

}

free(temp);

}

void display\_queue(void) {

struct node\* current = queue\_front;

if (queue\_front == NULL) {

printf("Очередь пуста\n");

return;

}

printf("\n=== ОЧЕРЕДЬ (FIFO) ===\n");

printf("Данные\n");

printf("------\n");

while (current != NULL) {

printf("%s\n", current->inf);

current = current->next;

}

}

void destroy\_queue(void) {

struct node\* current = queue\_front;

struct node\* next;

int count = 0;

while (current != NULL) {

next = current->next;

free(current);

current = next;

count++;

}

queue\_front = queue\_rear = NULL;

printf("Очередь очищена. Удалено элементов: %d\n", count);

}

// ==================== СТЕК (LIFO) ====================

struct node\* get\_stack\_struct(void) {

struct node\* p = (struct node\*)malloc(sizeof(struct node));

char s[256];

if (p == NULL) {

printf("Ошибка при распределении памяти\n");

exit(1);

}

printf("Введите название объекта: ");

scanf\_s("%s", s, (unsigned)sizeof(s));

if (\*s == 0) {

printf("Запись не была произведена\n");

free(p);

return NULL;

}

strcpy\_s(p->inf, sizeof(p->inf), s);

p->next = NULL;

return p;

}

void push(void) {

struct node\* p = get\_stack\_struct();

if (p == NULL) return;

p->next = stack\_top;

stack\_top = p;

printf("Элемент '%s' добавлен в вершину стека\n", p->inf);

}

void pop(void) {

if (stack\_top == NULL) {

printf("Стек пуст\n");

return;

}

struct node\* temp = stack\_top;

printf("Удален элемент из вершины стека: '%s'\n", temp->inf);

stack\_top = stack\_top->next;

free(temp);

}

void display\_stack(void) {

struct node\* current = stack\_top;

if (stack\_top == NULL) {

printf("Стек пуст\n");

return;

}

printf("\n=== СТЕК (LIFO) ===\n");

printf("Данные\n");

printf("------\n");

while (current != NULL) {

printf("%s\n", current->inf);

current = current->next;

}

}

void destroy\_stack(void) {

struct node\* current = stack\_top;

struct node\* next;

int count = 0;

while (current != NULL) {

next = current->next;

free(current);

current = next;

count++;

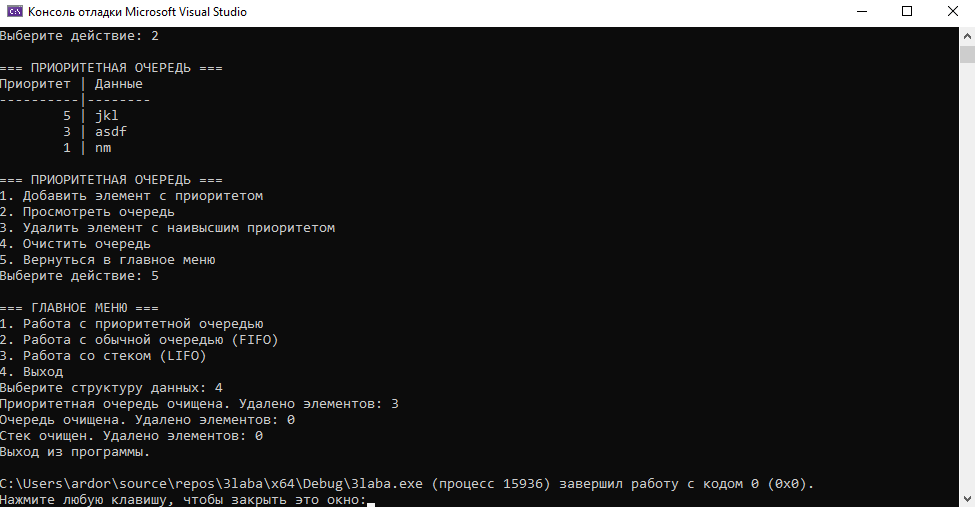
}

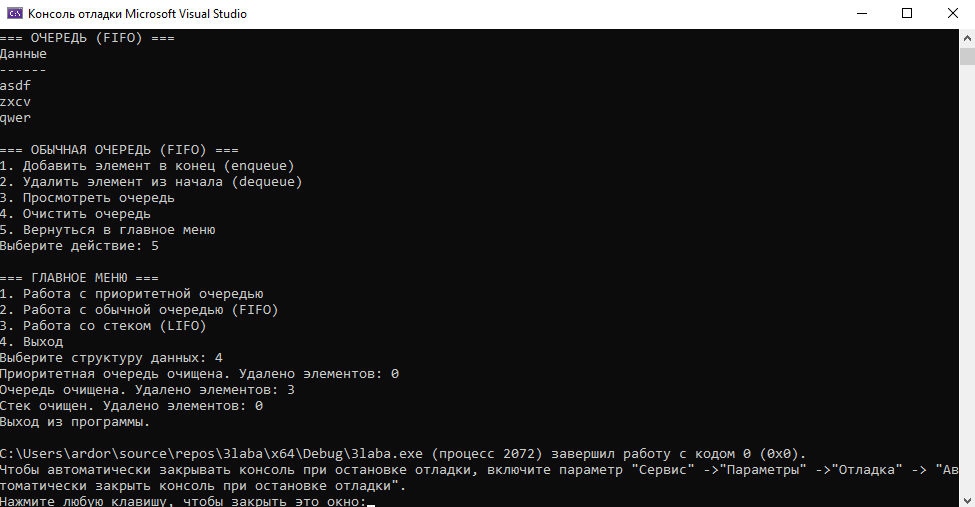
stack\_top = NULL;

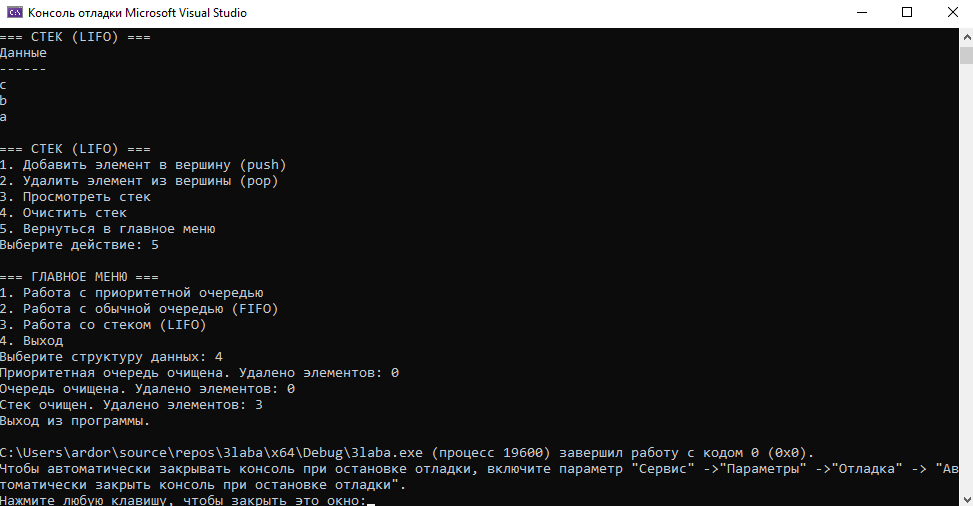
printf("Стек очищен. Удалено элементов: %d\n", count);

}

**Результат работы программы**







**Вывод**

Реализовали алгоритм, включающий в себя структуры приоритетной очереди, очереди и стека. Каждая структура данных работает независимо и демонстрирует соответствующий принцип обработки элементов. Научились реализовывать алгоритмы включающие в себя динамические списки и динамические структуры данных с эффективным распределением памяти.