МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»
(БГТУ им. В.Г. Шухова)**

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

Лабораторная работа №6

по дисциплине: Алгоритмы и структуры данных
по теме: «Структуры данных «стек» и «очередь» (Pascal/C)»

Выполнил: студент группы ПВ-223
Мелехов Артём Дмитриевич

Проверили:
асс. Солонченко Роман Евгеньевич

Белгород 2023 г.

Лабораторная работа №6. Структуры данных «стек» и «очередь» (C)

Цель работы: изучить СД типа «стек» и «очередь», научиться их программно реализовать и использовать.

Содержание отчета:

Решения заданий. Для каждого задания указаны:

– Название задания;

– Условие задания;

– Решение задания;

Вывод;

Полный листинг программы (с названиями директорий и файлов) и ссылка на репозиторий GitHub с выполненной работой.

Вариант №7.

Модуль для реализации стека: 7

Модуль для реализации очереди: 7

Задача: 7

Задание №1.

Для СД типа «стек» и «очередь» определить:

Пункт 1:

Абстрактный уровень представления СД.

Решение:

Стек – это абстрактная структура данных, организованная по принципу “последний пришел - первый ушел” (LIFO - Last In, First Out). Это означает, что последний элемент, добавленный в стек, будет первым, который будет удален.

Характер организованности и изменчивости:

Вершина стека: Это верхний элемент в стеке. Все операции (добавление и удаление элементов) происходят на вершине стека.

Основание стека: Это нижний элемент в стеке. Он добавляется первым и удаляется последним.

Набор допустимых операций:

`Stack *init_stack()`: Создает новый стек и возвращает указатель на него.
`void put_stack(Stack *stack, base_type data)`: Добавляет элемент на вершину стека.
`base_type get_stack(Stack *stack)`: Удаляет элемент с вершины стека и возвращает его.
`base_type read_stack(Stack *stack)`: Возвращает элемент с вершины стека без его удаления.
`bool empty_stack(Stack *stack)`: Проверяет, пуст ли стек.
`void done_stack(Stack *stack)`: Удаляет стек и освобождает выделенную под него память.

Очередь – это абстрактная структура данных, организованная по принципу “первый пришел - первый ушел” (FIFO - First In, First Out). Это означает, что первый элемент, добавленный в очередь, будет первым, который будет удален.

Характер организованности и изменчивости:

Голова очереди: Это первый элемент в очереди. Элементы удаляются из головы очереди.
Хвост очереди: Это последний элемент в очереди. Элементы добавляются в хвост очереди.

Набор допустимых операций:

`Queue *init_queue()`: Создает новую очередь и возвращает указатель на нее.
`void put_queue(Queue *queue, base_type data)`: Добавляет элемент в конец очереди.
`base_type get_queue(Queue *queue)`: Удаляет элемент из начала очереди и возвращает его.
`base_type read_queue(Queue *queue)`: Возвращает элемент из начала очереди без его удаления.
`bool empty_queue(Queue *queue)`: Проверяет, пуста ли очередь.
`void done_queue(Queue *queue)`: Удаляет очередь и освобождает выделенную под нее память.

Пункт 2:

Физический уровень представления СД.

Решение:

Стек:
Схема хранения: Стек может быть реализован с помощью массива или связанного списка. В коде стек реализован с помощью связанного списка, где каждый узел содержит данные и указатель на следующий узел.

Объём памяти, занимаемый экземпляром СД: Объём памяти, занимаемый стеком, зависит от количества элементов в стеке. Каждый узел в связанном списке занимает память для хранения данных и указателя на следующий узел. Структура `Stack` также содержит указатель `top` на вершину стека.

Формат внутреннего представления СД и способ его интерпретации: В коде каждый узел стека представлен структурой `Node`, содержащей данные (`data`) и указатель на следующий узел (`next`). Вершина стека (`top`) указывает на последний добавленный элемент.

Характеристика допустимых значений: Функции стека принимают и возвращают значения типа `base_type`. Это может быть любой тип данных, который определён как `base_type`.

Тип доступа к элементам: Стек предоставляет доступ только к верхнему элементу коллекции, что соответствует принципу LIFO. Можно использовать функцию `read_stack` для чтения верхнего элемента без его удаления и функцию `get_stack` для удаления верхнего элемента. Функция `put_stack` используется для добавления нового элемента на вершину стека.

Очередь:

Схема хранения: Очередь может быть реализована с помощью массива или связанного списка. В коде очередь реализована с помощью связанного списка, где каждый узел содержит данные и указатель на следующий узел.

Объём памяти, занимаемый экземпляром СД: Объём памяти, занимаемый очередью, зависит от количества элементов в очереди. Каждый узел в связанном списке занимает память для хранения данных и указателя на следующий узел. Структура `Queue` также содержит указатели `head` и `tail` на начало и конец очереди соответственно.

Формат внутреннего представления СД и способ его интерпретации: В коде каждый узел очереди представлен структурой `Node`, содержащей данные (`data`) и указатель на следующий узел (`next`). Начало и конец очереди (`head` и `tail`) указывают на первый и последний элементы в очереди соответственно.

Характеристика допустимых значений: Функции очереди принимают и возвращают значения типа `base_type`. Это может быть любой тип данных, который определён как `base_type`.

Тип доступа к элементам: Очередь предоставляет доступ только к первому элементу коллекции, что соответствует принципу FIFO. Можно использовать функцию `read_queue` для чтения первого элемента без его удаления и функцию `get_queue` для удаления первого элемента. Функция `put_queue` используется для добавления нового элемента в конец очереди.

Пункт 3:

Логический уровень представления СД.

Решение:

На логическом уровне стек представляет собой абстрактный тип данных (СД), который следует принципу “последний вошел, первый вышел” (LIFO). Это означает, что последний элемент, который был добавлен в стек, будет первым, который будет удален из него.

Реализация стека на языке С включает в себя следующие функции:

```
// Создает новый стек и возвращает указатель на него.
Stack *init_stack();

// Добавляет элемент на вершину стека.
void put_stack(Stack *stack, base_type data);

// Удаляет элемент с вершины стека и возвращает его.
base_type get_stack(Stack *stack);

// Возвращает элемент с вершины стека без его удаления.
base_type read_stack(Stack *stack);

// Проверяет, пуст ли стек.
bool empty_stack(Stack *stack);

// Удаляет стек и освобождает выделенную под него память.
void done_stack(Stack *stack);
```

Экземпляр стека представляет собой структуру `Stack`, которая содержит указатель `top` на вершину стека. Каждый элемент стека представлен структурой `Node`, которая содержит данные и указатель на следующий узел в стеке.

Реализация стека также включает в себя код обработки ошибок, который устанавливает глобальную переменную `stack_error` в случае возникновения ошибки. Это может быть полезно для отладки и обработки ошибок в коде.

На логическом уровне очередь представляет собой абстрактный тип данных (СД), который следует принципу “первый вошел, первый вышел” (FIFO). Это означает, что первый элемент, который был добавлен в очередь, будет первым, который будет удален из неё.

Реализация очереди на языке С включает в себя следующие функции:

```
// Создает новую очередь и возвращает указатель на нее.
Queue *init_queue();

// Добавляет элемент в конец очереди.
void put_queue(Queue *queue, base_type data);

// Удаляет элемент из начала очереди и возвращает его.
base_type get_queue(Queue *queue);

// Возвращает элемент из начала очереди без его удаления.
base_type read_queue(Queue *queue);

// Проверяет, пуста ли очередь.
bool empty_queue(Queue *queue);

// Удаляет очередь и освобождает выделенную под нее память.
void done_queue(Queue *queue);
```

Экземпляр очереди представляет собой структуру `Queue`, которая содержит указатели `head` и `tail` на начало и конец очереди соответственно. Каждый элемент очереди представлен структурой `Node`, которая содержит данные и указатель на следующий узел в очереди.

Реализация очереди также включает в себя код обработки ошибок, который устанавливает глобальную переменную `queue_error` в случае возникновения ошибки. Это может быть полезно для отладки и обработки ошибок в коде.

Задание №2.

Реализовать СД типа «стек» и «очередь» в соответствии с вариантом инди-видуального задания в виде модуля.

Решение:

Файл `data_structures/stack.h`:

```
// stack.h
#ifndef STACK_H
#define STACK_H
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#include "singly_linked_list.h"

typedef struct Stack {
    Node *top; // Указатель на вершину стека
} Stack;

extern const short stack_ok;
extern const short stack_not_mem; // Ошибка выделения памяти
extern const short stack_under;
short stack_error;

// Создает новый стек и возвращает указатель на него.
Stack *init_stack();

// Добавляет элемент на вершину стека.
void put_stack(Stack *stack, base_type data);

// Удаляет элемент с вершины стека и возвращает его.
base_type get_stack(Stack *stack);

// Возвращает элемент с вершины стека без его удаления.
base_type read_stack(Stack *stack);

// Проверяет, пуст ли стек.
bool empty_stack(Stack *stack);

// Удаляет стек и освобождает выделенную под него память.
void done_stack(Stack *stack);

#endif // STACK_H
```

Файл data_structures/stack.c:

```c
// stack.c

#include "stack.h"

const short stack_ok = 0;
const short stack_not_mem = 1; // Ошибка выделения памяти
const short stack_under = 2;
short stack_error = -1;
```

```
Stack *init_stack() {
    Stack *stack = (Stack *) malloc(sizeof(Stack));

    if (stack == NULL) {
        // Ошибка выделения памяти
        stack_error = stack_not_mem;

        return NULL;
    }

    stack->top = NULL;
    stack_error = stack_ok;

    return stack;
}

void put_stack(Stack *stack, base_type data) {
    Node *new_node = create_node(data);

    if (new_node == NULL)
        // Ошибка выделения памяти
        return;

    new_node->next = stack->top;
    stack->top = new_node;
}

base_type get_stack(Stack *stack) {
    if (stack->top == NULL) {
        // Стек пуст
        stack_error = stack_under;

        return -1; // Возвращаем -1 или другое значение, которое
не может быть в стеке
    }

    Node *temp = stack->top;
    base_type data = temp->data;

    stack->top = temp->next;

    free(temp);

    return data;
}
```

```
base_type read_stack(Stack *stack) {
    if (stack->top == NULL) {
        // Стек пуст
        stack_error = stack_under;

        return -1; // Возвращаем -1 или другое значение, которое
не может быть в стеке
    }

    return stack->top->data;
}

bool empty_stack(Stack *stack) {
    return (stack->top == NULL);
}

void done_stack(Stack *stack) {
    Node *current = stack->top;

    while (current != NULL) {
        Node *next = current->next;
        free(current);
        current = next;
    }

    free(stack);
}
```

Файл data_structures/queue.h:

```
// queue.h
#ifndef QUEUE_H
#define QUEUE_H

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

#include "singly_linked_list.h"

typedef struct Queue {
    Node *head; // Указатель на голову очереди
    Node *tail; // Указатель на хвост очереди
} Queue;

extern const short queue_ok;
extern const short queue_not_mem; // Ошибка выделения памяти
extern const short queue_under;
short queue_error;

// Создает новую очередь и возвращает указатель на нее.
Queue *init_queue();
```

```
// Добавляет элемент в конец очереди.
void put_queue(Queue *queue, base_type data);

// Удаляет элемент из начала очереди и возвращает его.
base_type get_queue(Queue *queue);

// Возвращает элемент из начала очереди без его удаления.
base_type read_queue(Queue *queue);

// Проверяет, пуста ли очередь.
bool empty_queue(Queue *queue);

// Удаляет очередь и освобождает выделенную под нее память.
void done_queue(Queue *queue);

#endif // QUEUE_H
```

Файл data_structures/queue.c:

```
// queue.c

#include "queue.h"

const short queue_ok = 0;
const short queue_not_mem = 1; // Ошибка выделения памяти
const short queue_under = 2;
short queue_error = -1;

Queue *init_queue() {
    Queue *queue = (Queue *) malloc(sizeof(Queue));

    if (queue == NULL) {
        // Ошибка выделения памяти
        queue_error = queue_not_mem;

        return NULL;
    }

    queue->head = NULL;
    queue->tail = NULL;
    queue_error = queue_ok;

    return queue;
}
```

```
void put_queue(Queue *queue, base_type data) {
    Node *new_node = create_node(data);

    if (new_node == NULL)
        // Ошибка выделения памяти
        return;

    if (queue->head == NULL) {
        queue->head = new_node;
        queue->tail = new_node;
    } else {
        queue->tail->next = new_node;
        queue->tail = new_node;
    }
}

base_type get_queue(Queue *queue) {
    if (queue->head == NULL) {
        // Очередь пуста
        queue_error = queue_under;

        return -1; // Возвращаем -1 или другое значение, которое
не может быть в очереди
    }

    Node *temp = queue->head;
    base_type data = temp->data;

    if (queue->head == queue->tail) {
        // В очереди был только один элемент
        queue->head = NULL;
        queue->tail = NULL;
    } else
        queue->head = temp->next;

    free(temp);

    return data;
}

base_type read_queue(Queue *queue) {
    if (queue->head == NULL) {
        // Очередь пуста
        queue_error = queue_under;

        return -1; // Возвращаем -1 или другое значение, которое
не может быть в очереди
    }

    return queue->head->data;
}
```

```
bool empty_queue(Queue *queue) {
    return (queue->head == NULL);
}

void done_queue(Queue *queue) {
    Node *current = queue->head;

    while (current != NULL) {
        Node *next = current->next;
        free(current);
        current = next;
    }

    free(queue);
}
```

Задание №3.

Разработать программу, моделирующую вычислительную систему с постоянным шагом по времени (дискретное время) в соответствии с вариантом индивидуального задания с использованием модуля, полученного в результате выполнения пункта 2. Результат работы программы представить в виде таблицы. В первом столбце указывается время моделирования 0, 1, 2, …, N. Во втором — для каждого момента времени указываются имена объектов (очереди — F1, F2, …, FN; стеки — S1, S2, …, SM; процессоры — P1, P2, …, PK), а в третьем — задачи (имя, время), находящиеся в объектах.

Решение:

Файл `lab6/lab6.h`:

```
//
// Created by Artyom on 08.11.2023.
//

#ifndef ALGORITMS_AND_DS_LAB6_H
#define ALGORITMS_AND_DS_LAB6_H

#include <string.h>

#include "../data_structure/stack.h"
#include "../data_structure/queue.h"

typedef struct TInquiry {
    char Name[10]; // имя запроса
    unsigned Time; // время обслуживания
    char T; // тип задачи: 1 — T1, 2 — T2, 3 — T3
} TInquiry;

typedef struct Processor {
    TInquiry *task;
    unsigned time_left;
```

```
} Processor;
/*
 * Система состоит из двух процессоров P1 и P2, трех очередей
F1, F2, F3 и стека. В систему
 * могут поступать запросы на выполнение задач трех типов — T1,
T2, T3. Задача типа T1 может
 * выполняться только процессором P1. Задача типа T2 может
выполняться только процессором P2.
 * Задача типа T3 может выполняться любым процессором. Запрос
можно представить записью TInquiry.
 * Поступающие запросы ставятся в соответствующие типам задач
очереди. Если очередь F1 не пуста
 * и процессор P1 свободен, то задача из очереди F1 поступает на
обработку в процессор P1. Если
 * процессор P1 обрабатывает задачу типа T3, а процессор P2
свободен и очередь F2 пуста, то задача
 * из процессора P1 поступает в процессор P2, а задача из
очереди F1 в процессор P1, если же процессор
 * P2 занят или очередь F2 не пуста, то задача из процессора P1
помещается в стек.
 * Если очередь F2 не пуста и процессор P2 свободен, то задача
из очереди F2 поступает на обработку
 * в процессор P2. Если процессор P2 обрабатывает задачу типа
T3, а процессор P1 свободен и очередь
 * F1 пуста, то задача из процессора P2 поступает в процессор
P1, а задача из очереди F2 — в процессор
 * P2, если же процессор P1 занят или очередь F1 не пуста, то
задача из процессора P1 помещается в стек.
 * Если очередь F3 не пуста и процессор P1 свободен, и очередь
F1 пуста или свободен процессор P2
 * и очередь F2 пуста, то задача из очереди F3 поступает на
обработку в свободный процессор. Задача
 * из стека поступает на обработку в свободный процессор P1,
если очередь F1 пуста, или в
 * свободный процессор P2, если очередь F2 пуста.
 */
void simulate(unsigned N);

#endif //ALGORITMS_AND_DS_LAB6_H
```

Файл lab6/lab6.h:

```
//
// Created by Artyom on 08.11.2023.
//

#include "lab6.h"
```

```
Processor *init_processor() {
    Processor *processor = (Processor *)
malloc(sizeof(Processor));

    if (processor == NULL)
        // Ошибка выделения памяти
        return NULL;

    processor->task = NULL;
    processor->time_left = 0;

    return processor;
}

void put_processor(Processor *processor, TInquiry *task) {
    processor->task = task;
    processor->time_left = task->Time;
}

TInquiry *get_processor(Processor *processor) {
    TInquiry *task = processor->task;

    processor->task = NULL;
    processor->time_left = 0;

    return task;
}

bool empty_processor(Processor *processor) {
    return (processor->task == NULL);
}

void done_processor(Processor *processor) {
    free(processor);
}
```

```
// Основная функция моделирования
void simulate(unsigned N) {
    Queue *F1 = init_queue();
    Queue *F2 = init_queue();
    Queue *F3 = init_queue();
    Stack *S = init_stack();
    Processor *P1 = init_processor();
    Processor *P2 = init_processor();

    // Заполнение очередей F1, F2 и F3 запросами
    TInquiry *task1 = (TInquiry *) malloc(sizeof(TInquiry));
    strcpy(task1->Name, "Task 1");
    task1->Time = 5;
    task1->T = '1';
    put_queue(F1, (base_type) task1);

    TInquiry *task2 = (TInquiry *) malloc(sizeof(TInquiry));
    strcpy(task2->Name, "Task 2");
    task2->Time = 3;
    task2->T = '2';
    put_queue(F2, (base_type) task2);

    TInquiry *task3 = (TInquiry *) malloc(sizeof(TInquiry));
    strcpy(task3->Name, "Task 3");
    task3->Time = 4;
    task3->T = '3';
    put_queue(F3, (base_type) task3);

    for (unsigned t = 0; t <= N; t++) {
        printf("Time: %u\n", t);

        // Обработка запросов
        if (!empty_queue(F1) && empty_processor(P1))
            put_processor(P1, (TInquiry *) get_queue(F1));
        else if (!empty_processor(P1) && P1->task->T == '3' &&
empty_processor(P2) &&
                 empty_queue(F2)) {
            put_processor(P2, get_processor(P1));

            if (!empty_queue(F1))
                put_processor(P1, (TInquiry *) get_queue(F1));
            else
                put_stack(S, (base_type) get_processor(P1));
        }

        if (!empty_queue(F2) && empty_processor(P2))
            put_processor(P2, (TInquiry *) get_queue(F2));
        else if (!empty_processor(P2) && P2->task->T == '3' &&
empty_processor(P1) &&
                 empty_queue(F1)) {
            put_processor(P1, get_processor(P2));

            if (!empty_queue(F2))
```

```
                put_processor(P2, (TInquiry *) get_queue(F2));
            else
                put_stack(S, (base_type) get_processor(P2));
        }

        if (!empty_queue(F3) && (empty_processor(P1) &&
empty_queue(F1) ||
                                 empty_processor(P2) &&
empty_queue(F2))) {
            if (empty_processor(P1) && empty_queue(F1))
                put_processor(P1, (TInquiry *) get_queue(F3));
            else
                put_processor(P2, (TInquiry *) get_queue(F3));
        }

        if (!empty_stack(S) && (empty_processor(P1) &&
empty_queue(F1) ||
                                empty_processor(P2) &&
empty_queue(F2))) {
            if (empty_processor(P1) && empty_queue(F1))
                put_processor(P1, (TInquiry *) get_stack(S));
            else
                put_processor(P2, (TInquiry *) get_stack(S));
        }

        if (!empty_processor(P1)) {
            P1->time_left--;

            if (!P1->time_left)
                free(get_processor(P1));
        }

        if (!empty_processor(P2)) {
            P2->time_left--;

            if (!P2->time_left)
                free(get_processor(P2));
        }

        printf("\n");
    }

    done_processor(P1);
    done_processor(P2);
    done_queue(F1);
    done_queue(F2);
    done_queue(F3);
    done_stack(S);
}
```

Файл main.c:

```c
#include "lab6/lab6.h"

typedef long long base_type; // Замените на нужный тип данных

// Здесь происходит запуск последней выполненной лабораторной работы
int main() {
    unsigned N = 10; // Задайте здесь количество шагов моделирования
    simulate(N);

    return 0;
}
```

Вывод: в ходе выполнения лабораторной работы изучены и программно реализованы СД типа «стек» и «очередь».

Ссылка на GitHub (lab5): https://github.com/SStaryi/algorithms_and_DS

Полный листинг программы:

Файл data_structure/singly_linked_list.h:

```c
// singly_linked_list.h
#ifndef SINGLY_LINKED_LIST_H
#define SINGLY_LINKED_LIST_H

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

typedef long long base_type; // Замените на нужный тип данных

typedef struct Node {
    base_type data; // Данные для хранения в узле
    struct Node *next; // Указатель на следующий узел
} Node;

extern const short list_ok_singly;
extern const short list_not_mem_singly; // Ошибка выделения памяти
extern const short list_under_singly;
extern const short list_end_singly;
extern short list_error_singly;

// Создает новый узел с заданными данными и возвращает указатель на него.
Node *create_node(base_type data);
```

```
// Вставляет новый узел с заданными данными в конец списка.
void insert_singly(Node **head, base_type data);

// Удаляет первый узел с заданными данными из списка.
void delete(Node **head, base_type data);

// Возвращает указатель на первый узел с заданными данными.
Node *find(Node *head, base_type data);

// Выводит все элементы списка.
void print_list(Node *head);

#endif // SINGLY_LINKED_LIST_H
```

Файл data_structure/singly_linked_list.c:

```
// singly_linked_list.c

#include "singly_linked_list.h"

const short list_ok_singly = 0;
const short list_not_mem_singly = 1; // Ошибка выделения памяти
const short list_under_singly = 2;
const short list_end_singly = 3;
short list_error_singly = -1;

Node *create_node(base_type data) {
    Node *new_node = (Node *) malloc(sizeof(Node));

    if (new_node == NULL) {
        // Ошибка выделения памяти
        list_error_singly = list_not_mem_singly;

        return NULL;
    }

    new_node->data = data;
    new_node->next = NULL;
    list_error_singly = list_ok_singly;

    return new_node;
}
```

```
void insert_singly(Node **head, base_type data) {
    Node *new_node = create_node(data);

    if (new_node == NULL)
        // Ошибка выделения памяти
        return;

    if (*head == NULL) {
        *head = new_node;

        return;
    }

    Node *current = *head;

    while (current->next != NULL)
        current = current->next;

    current->next = new_node;
}

void delete(Node **head, base_type data) {
    if (*head == NULL) {
        // Список пуст
        list_error_singly = list_under_singly;

        return;
    }

    Node *current = *head;
    Node *previous = NULL;

    while (current != NULL && current->data != data) {
        previous = current;
        current = current->next;
    }

    if (current == NULL) {
        // Элемент не найден в списке
        list_error_singly = list_end_singly;

        return;
    }

    if (previous == NULL)
        // Элемент находится в начале списка
        *head = current->next;
    else
        previous->next = current->next;

    free(current);
}
```

```c
Node *find(Node *head, base_type data) {
    Node *current = head;

    while (current != NULL && current->data != data)
        current = current->next;

    if (current == NULL) {
        // Элемент не найден в списке
        list_error_singly = list_end_singly;

        return NULL;
    }

    return current;
}

void print_list(Node *head) {
    Node *current = head;

    while (current != NULL) {
        printf("%lld ", current->data);

        current = current->next;
    }

    printf("\n");
}
```

Файл data_structure/stack.h:

```c
// stack.h
#ifndef STACK_H
#define STACK_H

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

#include "singly_linked_list.h"

typedef struct Stack {
    Node *top; // Указатель на вершину стека
} Stack;

extern const short list_ok_stack;
extern const short list_not_mem_stack; // Ошибка выделения
памяти
extern const short list_under_stack;
extern short list_error_stack;

// Создает новый стек и возвращает указатель на него.
Stack *init_stack();
```

```
// Добавляет элемент на вершину стека.
void put_stack(Stack *stack, base_type data);

// Удаляет элемент с вершины стека и возвращает его.
base_type get_stack(Stack *stack);

// Возвращает элемент с вершины стека без его удаления.
base_type read_stack(Stack *stack);

// Проверяет, пуст ли стек.
bool empty_stack(Stack *stack);

// Удаляет стек и освобождает выделенную под него память.
void done_stack(Stack *stack);

#endif // STACK_H
```

Файл data_structure/stack.c:

```
// stack.c

#include "stack.h"

const short list_ok_stack = 0;
const short list_not_mem_stack = 1; // Ошибка выделения памяти
const short list_under_stack = 2;
short list_error_stack = -1;

Stack *init_stack() {
    Stack *stack = (Stack *) malloc(sizeof(Stack));

    if (stack == NULL) {
        // Ошибка выделения памяти
        list_error_stack = list_not_mem_stack;

        return NULL;
    }

    stack->top = NULL;
    list_error_stack = list_ok_stack;

    return stack;
}
```

```
void put_stack(Stack *stack, base_type data) {
    Node *new_node = create_node(data);

    if (new_node == NULL)
        // Ошибка выделения памяти
        return;

    new_node->next = stack->top;
    stack->top = new_node;
}

base_type get_stack(Stack *stack) {
    if (stack->top == NULL) {
        // Стек пуст
        list_error_stack = list_under_stack;

        return -1; // Возвращаем -1 или другое значение, которое
не может быть в стеке
    }

    Node *temp = stack->top;
    base_type data = temp->data;

    stack->top = temp->next;

    free(temp);

    return data;
}

base_type read_stack(Stack *stack) {
    if (stack->top == NULL) {
        // Стек пуст
        list_error_stack = list_under_stack;

        return -1; // Возвращаем -1 или другое значение, которое
не может быть в стеке
    }

    return stack->top->data;
}

bool empty_stack(Stack *stack) {
    return (stack->top == NULL);
}
```

```
void done_stack(Stack *stack) {
    Node *current = stack->top;

    while (current != NULL) {
        Node *next = current->next;
        free(current);
        current = next;
    }

    free(stack);
}
```

Файл data_structure/queue.h:

```
// queue.h
#ifndef QUEUE_H
#define QUEUE_H

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

#include "singly_linked_list.h"

typedef struct Queue {
    Node *head; // Указатель на голову очереди
    Node *tail; // Указатель на хвост очереди
} Queue;

extern const short list_ok_queue;
extern const short list_not_mem_queue; // Ошибка выделения
памяти
extern const short list_under_queue;
extern short list_error_queue;

// Создает новую очередь и возвращает указатель на нее.
Queue *init_queue();

// Добавляет элемент в конец очереди.
void put_queue(Queue *queue, base_type data);

// Удаляет элемент из начала очереди и возвращает его.
base_type get_queue(Queue *queue);

// Возвращает элемент из начала очереди без его удаления.
base_type read_queue(Queue *queue);

// Проверяет, пуста ли очередь.
bool empty_queue(Queue *queue);
```

```
// Удаляет очередь и освобождает выделенную под нее память.
void done_queue(Queue *queue);

#endif // QUEUE_H
```

Файл data_structure/queue.c:

```
// queue.c

#include "queue.h"

const short list_ok_queue = 0;
const short list_not_mem_queue = 1; // Ошибка выделения памяти
const short list_under_queue = 2;
short list_error_queue = -1;

Queue *init_queue() {
    Queue *queue = (Queue *) malloc(sizeof(Queue));

    if (queue == NULL) {
        // Ошибка выделения памяти
        list_error_queue = list_not_mem_queue;

        return NULL;
    }

    queue->head = NULL;
    queue->tail = NULL;
    list_error_queue = list_ok_queue;

    return queue;
}

void put_queue(Queue *queue, base_type data) {
    Node *new_node = create_node(data);

    if (new_node == NULL)
        // Ошибка выделения памяти
        return;

    if (queue->head == NULL) {
        queue->head = new_node;
        queue->tail = new_node;
    } else {
        queue->tail->next = new_node;
        queue->tail = new_node;
    }
}
```

```
base_type get_queue(Queue *queue) {
    if (queue->head == NULL) {
        // Очередь пуста
        list_error_queue = list_under_queue;

        return -1; // Возвращаем -1 или другое значение, которое
не может быть в очереди
    }

    Node *temp = queue->head;
    base_type data = temp->data;

    if (queue->head == queue->tail) {
        // В очереди был только один элемент
        queue->head = NULL;
        queue->tail = NULL;
    } else
        queue->head = temp->next;

    free(temp);

    return data;
}

base_type read_queue(Queue *queue) {
    if (queue->head == NULL) {
        // Очередь пуста
        list_error_queue = list_under_queue;

        return -1; // Возвращаем -1 или другое значение, которое
не может быть в очереди
    }

    return queue->head->data;
}

bool empty_queue(Queue *queue) {
    return (queue->head == NULL);
}

void done_queue(Queue *queue) {
    Node *current = queue->head;

    while (current != NULL) {
        Node *next = current->next;
        free(current);
        current = next;
    }

    free(queue);
}
```

Файл lab6/lab6.h:

```
//
// Created by Artyom on 08.11.2023.
//

#ifndef ALGORITMS_AND_DS_LAB6_H
#define ALGORITMS_AND_DS_LAB6_H

#include <string.h>

#include "../data_structure/stack.h"
#include "../data_structure/queue.h"

typedef struct TInquiry {
    char Name[10]; // имя запроса
    unsigned Time; // время обслуживания
    char T; // тип задачи: 1 — T1, 2 — T2, 3 — T3
} TInquiry;

typedef struct Processor {
    TInquiry *task;
    unsigned time_left;
} Processor;
```

```c
/*
 * Система состоит из двух процессоров P1 и P2, трех очередей
F1, F2, F3 и стека. В систему
 * могут поступать запросы на выполнение задач трех типов — T1,
T2, T3. Задача типа T1 может
 * выполняться только процессором P1. Задача типа T2 может
выполняться только процессором P2.
 * Задача типа T3 может выполняться любым процессором. Запрос
можно представить записью TInquiry.
 * Поступающие запросы ставятся в соответствующие типам задач
очереди. Если очередь F1 не пуста
 * и процессор P1 свободен, то задача из очереди F1 поступает на
обработку в процессор P1. Если
 * процессор P1 обрабатывает задачу типа T3, а процессор P2
свободен и очередь F2 пуста, то задача
 * из процессора P1 поступает в процессор P2, а задача из
очереди F1 в процессор P1, если же процессор
 * P2 занят или очередь F2 не пуста, то задача из процессора P1
помещается в стек.
 * Если очередь F2 не пуста и процессор P2 свободен, то задача
из очереди F2 поступает на обработку
 * в процессор P2. Если процессор P2 обрабатывает задачу типа
T3, а процессор P1 свободен и очередь
 * F1 пуста, то задача из процессора P2 поступает в процессор
P1, а задача из очереди F2 — в процессор
 * P2, если же процессор P1 занят или очередь F1 не пуста, то
задача из процессора P1 помещается в стек.
 * Если очередь F3 не пуста и процессор P1 свободен, и очередь
F1 пуста или свободен процессор P2
 * и очередь F2 пуста, то задача из очереди F3 поступает на
обработку в свободный процессор. Задача
 * из стека поступает на обработку в свободный процессор P1,
если очередь F1 пуста, или в
 * свободный процессор P2, если очередь F2 пуста.
 */
void simulate(unsigned N);

#endif //ALGORITMS_AND_DS_LAB6_H
```

Файл lab6/lab6.c:

```c
//
// Created by Artyom on 08.11.2023.
//

#include "lab6.h"
```

```
Processor *init_processor() {
    Processor *processor = (Processor *)
malloc(sizeof(Processor));

    if (processor == NULL)
        // Ошибка выделения памяти
        return NULL;

    processor->task = NULL;
    processor->time_left = 0;

    return processor;
}

void put_processor(Processor *processor, TInquiry *task) {
    processor->task = task;
    processor->time_left = task->Time;
}

TInquiry *get_processor(Processor *processor) {
    TInquiry *task = processor->task;

    processor->task = NULL;
    processor->time_left = 0;

    return task;
}

bool empty_processor(Processor *processor) {
    return (processor->task == NULL);
}

void done_processor(Processor *processor) {
    free(processor);
}
```

```
// Основная функция моделирования
void simulate(unsigned N) {
    Queue *F1 = init_queue();
    Queue *F2 = init_queue();
    Queue *F3 = init_queue();
    Stack *S = init_stack();
    Processor *P1 = init_processor();
    Processor *P2 = init_processor();

    // Заполнение очередей F1, F2 и F3 запросами
    TInquiry *task1 = (TInquiry *) malloc(sizeof(TInquiry));
    strcpy(task1->Name, "Task 1");
    task1->Time = 5;
    task1->T = '1';
    put_queue(F1, (base_type) task1);

    TInquiry *task2 = (TInquiry *) malloc(sizeof(TInquiry));
    strcpy(task2->Name, "Task 2");
    task2->Time = 3;
    task2->T = '2';
    put_queue(F2, (base_type) task2);

    TInquiry *task3 = (TInquiry *) malloc(sizeof(TInquiry));
    strcpy(task3->Name, "Task 3");
    task3->Time = 4;
    task3->T = '3';
    put_queue(F3, (base_type) task3);

    for (unsigned t = 0; t <= N; t++) {
        printf("Time: %u\n", t);

        // Обработка запросов
        if (!empty_queue(F1) && empty_processor(P1))
            put_processor(P1, (TInquiry *) get_queue(F1));
        else if (!empty_processor(P1) && P1->task->T == '3' &&
empty_processor(P2) &&
                 empty_queue(F2)) {
            put_processor(P2, get_processor(P1));

            if (!empty_queue(F1))
                put_processor(P1, (TInquiry *) get_queue(F1));
            else
                put_stack(S, (base_type) get_processor(P1));
        }

        if (!empty_queue(F2) && empty_processor(P2))
            put_processor(P2, (TInquiry *) get_queue(F2));
        else if (!empty_processor(P2) && P2->task->T == '3' &&
empty_processor(P1) &&
                 empty_queue(F1)) {
            put_processor(P1, get_processor(P2));

            if (!empty_queue(F2))
```

```
                put_processor(P2, (TInquiry *) get_queue(F2));
            else
                put_stack(S, (base_type) get_processor(P2));
        }

        if (!empty_queue(F3) && (empty_processor(P1) &&
empty_queue(F1) ||
                                 empty_processor(P2) &&
empty_queue(F2))) {
            if (empty_processor(P1) && empty_queue(F1))
                put_processor(P1, (TInquiry *) get_queue(F3));
            else
                put_processor(P2, (TInquiry *) get_queue(F3));
        }

        if (!empty_stack(S) && (empty_processor(P1) &&
empty_queue(F1) ||
                                empty_processor(P2) &&
empty_queue(F2))) {
            if (empty_processor(P1) && empty_queue(F1))
                put_processor(P1, (TInquiry *) get_stack(S));
            else
                put_processor(P2, (TInquiry *) get_stack(S));
        }

        if (!empty_processor(P1)) {
            P1->time_left--;

            if (!P1->time_left)
                free(get_processor(P1));
        }

        if (!empty_processor(P2)) {
            P2->time_left--;

            if (!P2->time_left)
                free(get_processor(P2));
        }

        printf("\n");
    }

    done_processor(P1);
    done_processor(P2);
    done_queue(F1);
    done_queue(F2);
    done_queue(F3);
    done_stack(S);
}
```

Файл main.c:

```c
#include "lab6/lab6.h"

typedef long long base_type; // Замените на нужный тип данных

// Здесь происходит запуск последней выполненной лабораторной работы
int main() {
    unsigned N = 10; // Задайте здесь количество шагов моделирования
    simulate(N);

    return 0;
}
```