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» (C)»**

Выполнил/а: ст. группы ПВ-231
Чупахина София Александровна
Проверил:
Акиньшин Даниил Иванович

Белгород, 2024

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

Задания:

1. Для СД типа «стек» и «очередь» определить:
   (a) Абстрактный уровень представления СД:
      i. Характер организованности и изменчивости,
      ii. Набор допустимых операций.
   (b) Физический уровень представления СД:
      i. Схему хранения;
      ii. Объем памяти, занимаемый экземпляром СД;
      iii. Формат внутреннего представления СД и способ его интерпретации;
      iv. Характеристику допустимых значений;
      v. Тип доступа к элементам.
   (c) Логический уровень представления СД. Способ описания СД и экземпляра СД на языке программирования.
2. Реализовать СД типа «стек» и «очередь» в соответствии с вариантом индивидуального задания в виде модуля;
3. Разработать программу, моделирующую вычислительную систему с постоянным шагом по времени (дискретное время) в соответствии с вариантом индивидуального задания (табл.16) с использованием модуля, полученного в результате выполнения пункта 2. Результат работы программы представить в виде таблицы 15. В первом столбце указывается время моделирования $0, 1, 2, \ldots, N$. Во втором — для каждого момента времени указываются имена объектов (очереди — $F_1, F_2, \ldots, F_N$; стеки — $S_1, S_2, \ldots, S_M$; процессоры — $P_1, P_2, \ldots, P_K$), а в третьем — задачи (имя, время), находящиеся в объектах.

Содержание

Задание 1:

Опишем СД типа «стек» и «очередь».

Абстрактный уровень:

Задание 1.1.1:

Как СД типа «стек», так и СД типа «очередь» являются линейными СД. Как и рассматриваемые ранее СД типа «массив» и «линейный список», они отображают некую упорядоченную последовательность элементов, и для каждого из них, кроме первого и последнего, существует предыдущий и следующий. СД типа «стек» и «очередь» могут быть реализованы для любого типа элементов. Специфика СД типа «стек» и «очередь» в том, что для них невозможна перезапись и чтение элемента по произвольному индексу (как для массива), а также включение и исключение элемента на произвольную позицию (как для линейных списков). В СД типа стек включать, исключать и считывать элемент можно только с одной определенной позиции, которая называется вершиной стека. Таким образом, последний включенный элемент будет ближе всего к вершине стека, из которой проводится исключение, и потому при исключении он будет извлечен первым. Это свойство дало стеку сокращенное название LIFO (от англ. Last In, First Out). В СД типа очередь элементы всегда включаются с одной позиции, которая называется хвостом очереди, и исключаются и считываются с другой позиции, которая называется головой очереди. Только что включенный с хвоста элемент отделяется от головы всеми теми элементами, которые были включены в список до него. По аналогии со стеком, очередь имеет указывающее на это свойство сокращенное название FIFO (от англ. First In, First Out).

Обычно роль вершины, головы или хвоста отдается первому или последнему элементу массива.

«Стек» и «очередь» являются динамическими СД: несмотря на невозможность перезаписи и осуществление включения и исключения только с одной позиции (эти позиции совпадают для стека и не совпадают для очереди) в нем могут меняться как сами элементы, так и их количество. Количество элементов ограничено только способом реализации структуры; этот аспект подробнее рассмотрим в задании 1.2.

Задание 1.1.2:

СД типа «стек» и «очередь» не реализованы в языках программирования по умолчанию, то есть являются производными структурами данных. Тем не менее, для каждой из них существуют операции, которые должны быть реализованы по определению. Для СД типа «стек» это операции:

1. Инициализация.
2. Включение элемента.
3. Исключение элемента.
4. Чтение элемента.

5. Проверка пустоты стека.

6. Уничтожение стека.

А для СД типа «очередь» — следующие операции:

1. Инициализация.

2. Включение элемента.

3. Исключение элемента.

4. Чтение элемента (с головы очереди).

5. Проверка пустоты очереди.

6. Уничтожение очереди.

Физический уровень:

Задание 1.2.1:

Для СД типа «стек» и «очередь» может использоваться как прямоугольная, так и связная схема хранения. При использовании прямоугольной схемы для хранения элементов стека или очереди используется массив; тогда последовательности битов, представляющие нулевой, первый, второй и так далее элемент, идут друг за другом по возрастанию индекса подряд, без разделителей. Точно так же для реализации стека или очереди может использоваться связная схема хранения — тогда элементы этих СД будут храниться в связном списке. В таком случае значение каждого элемента включено в структуру типа «запись», которая хранит также указатель на следующий элемент (для односвязного списка) или указатели на следующий и на предыдущий элемент (для двусвязного списка). В любом случае, над СД типа «стек» и «очередь» реализуются также и функции, ограничивающие доступ к элементами массива или списка по умолчанию и обеспечивающие выполнение ключевых условий: включение, исключение или чтение элемента только с вершины стека; включение элемента только с хвоста и исключение и чтение только с головы для очереди.

Задание 1.2.2:

Количество памяти, занимаемой СД типа «стек» или «очередь», зависит от того, какое количество памяти занимает его базовый тип, есть ли необходимость хранить дополнительную информацию (указатели на другие элементы для связной схемы хранения) о каждом элементе, и сколько элементов содержит СД. Если базовый тип занимает T байт, для каждого элемента необходимо также хранить P (P также может быть равно 0) указателей типа *unsigned long*, занимающего 8 байт, а список содержит K элементов, то количество памяти, занимаемой экземпляром любой из этих СД, будет равно $(T + P) * K$ байт.

Задание 1.2.3:

Если СД типа «стек» или «очередь» с длиной K использует последовательную схему хранения, то значение этой структуры хранится как K идущих подряд последовательностей битов, кодирующих значения элементов СД в соответствии с правилами для данного базового типа. Для связных же вариантов реализации каждый узел (структура с одним полем «значение элемента» и несколькими полями «указатели») хранится независимо друг от друга, и значения в полях узла также могут храниться в памяти компьютера не подряд (либо в порядке, отличном от заданного). Значение элемента переводится в двоичный код в зависимости от базового типа; указатели кодируются как положительные целые числа — переводятся в двоичную систему счисления и дополняются нулями слева до размера в 8 байт.

Задание 1.2.4:

Диапазоны допустимых значений СД типа «стек» и «очередь» связаны с диапазонами допустимых значений их базовых элементов. Если рассматривать стек или очередь некоторой фиксированной длины K, то количество возможных значений для экземпляров этих СД равно возведенному в степень K количеству возможных значений для их базового элемента. Соответственно, для СД типа «стек» или «очередь» количество возможных значений равно сумме количеств возможных значений этих СД с длинами от 0 до max, где max — максимальная длина стека или очереди (однозначно определенная в случаях, когда эти СД реализованы на массиве). Обозначая количество допустимых значений как CAR (кардинальное число), получим формулу $CAR(STACK) = CAR(FIFO) = CAR(BaseType)^0 + CAR(BaseType)^1 + \ldots + CAR(BaseType)^max$.

Задание 1.2.5:

СД типа «стек» и «очередь» имеют ограниченный доступ к элементам. Для стека чтение осуществляется только с его вершины, для очереди — только с ее головы. Соответственно, чтобы получить некоторый (не вершинный) элемент стека, нужно исключить из него все элементы, которые отделяют его от вершины, а чтобы получить некоторый (не головной) элемент очереди, нужно исключить из нее все элементы, которые отделяют его от головы. С этой точки зрения логичнее будет сказать, что доступ к элементам в этих СД последователен.

Логический уровень:

СД типа «стек» и «очередь» не являются встроенными, и потому сначала необходимо реализовать их, выбрав один из способов отображения. Только после этого их можно описать на логическом уровне (представить на языке программирования). Приведем здесь описания для той реализации, которая будет выполнена в задании 2 этой лабораторной работы. Стек отображается на последовательный линейный список, и его вершиной является первый элемент этого списка; кольцевая очередь отображается на массив в динамической памяти. Пока что мы не реализовали функции для работы с этими СД, и заполнение стека и очереди значениями, как и задание основных параметров, будет происходить более примитивными способами.

```
1 #include <malloc.h>
2 #include <memory.h>
```

```c


#define SizeList 100
typedef void *BaseType;
typedef unsigned ptrel;
typedef struct {
    BaseType MemList[SizeList];
    ptrel ptr;
    unsigned int N; // длина списка
    unsigned int Size;
} List;
typedef List Stack;

int main () {
    Stack st;
    st.Size = sizeof(int);
    for (int i = 0; i < SizeList; i++) {
        st.MemList[i] = malloc(st.Size);
    }
    int a = 3;
    int b = 7;
    int c = -12;
    memcpy(st.MemList[0], &a, st.Size);
    memcpy(st.MemList[1], &b, st.Size);
    memcpy(st.MemList[2], &c, st.Size);

    st.ptr = 1;
    st.N = 3;

    return 0;
}
```

../АСД 6 си/stack/example.c

```c
#include <malloc.h>
#include <memory.h>

typedef void *BaseType;
typedef struct {
    BaseType *Buf;
    unsigned SizeBuf; // Максимальная длина очереди
    unsigned SizeEl; // Размер элемента очереди
    unsigned Uk1; // Указатель на «голову» очереди
    unsigned Uk2; // Указатель на «хвост» очереди
    unsigned N;   // Количество элементов очереди
} Fifo;

int main () {
```

```
15      Fifo fi;
16      fi.SizeBuf = 10;
17      fi.Buf = malloc(fi.SizeBuf * sizeof(void*));
18      fi.SizeEl = sizeof(int);
19      for (int i = 0; i < fi.SizeBuf; i++) {
20          fi.Buf[i] = malloc(fi.SizeEl);
21      }
22      int a = 3;
23      int b = 7;
24      int c = -12;
25      memcpy(fi.Buf[3], &a, fi.SizeEl);
26      memcpy(fi.Buf[4], &b, fi.SizeEl);
27      memcpy(fi.Buf[5], &c, fi.SizeEl);
28
29      fi.Uk1 = 3;
30      fi.Uk2 = 6;
31      fi.N = 3;
32
33      return 0;
34  }
```

../АСД 6 си/fifo/example.c

Индивидуальное задание; вариант 21

Модуль 10 (для стека): Стек на ПЛС. Вершина стека – первый элемент ПЛС.

Реализация на языке С:

```
#if !defined(__STACK10_H)
#define __STACK10_H
#include "list6.h"// Смотреть лаб. раб. №5
const StackOk = ListOk;
const StackUnder = ListUnder;
const StackOver = ListNotMem;
int StackError; // Переменная ошибок
typedef List Stack;
void InitStack(Stack *s, unsigned Size); /* Инициализация стека */
void PutStack(Stack *s, void *E); // Поместить элемент в стек
void GetStack(Stack *s, void *E); // Извлечь элемент из стека
int EmptyStack(Stack s); // Проверка: стек пуст?
void ReadStack(Stack s, void *E); /* Прочитать элемент из вершины стека */
void DoneStack(Stack *s); // Уничтожить стек
#endif
```

Модуль 11 (для очереди): Очередь (кольцевая) на массиве в динамической памяти.

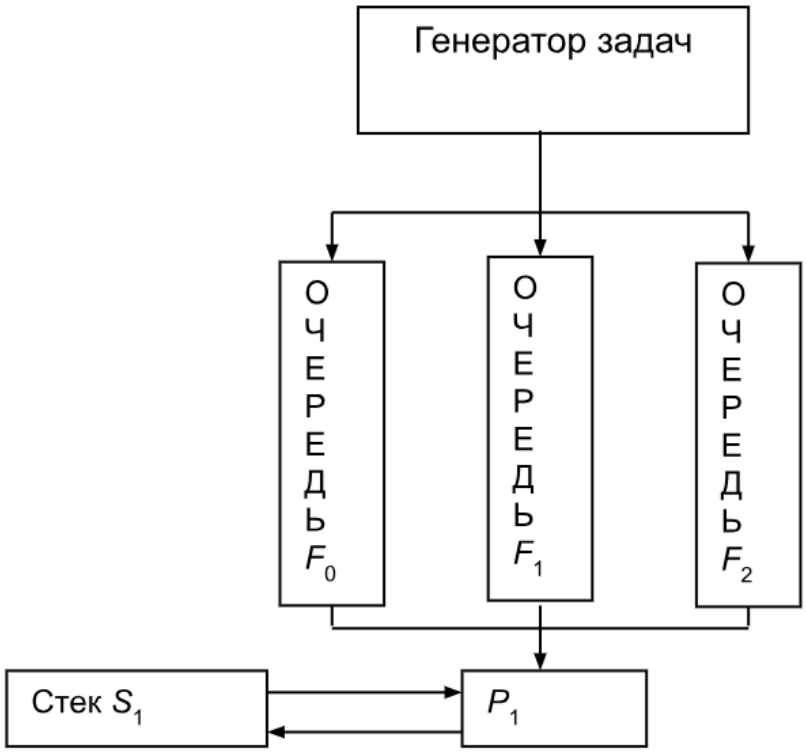
```
#if !defined(__FIFO11_H)
#define __FIFO11_H
const FifoOk = 0;
const FifoUnder = 1;
const FifoOver = 2;
int FifoError; // Переменная ошибок
typedef void *BaseType;
typedef struct Fifo
{
BaseType *Buf;
unsigned SizeBuf; // Максимальная длина очереди
unsigned SizeEl; // Размер элемента очереди
unsigned Uk1; // Указатель на «голову» очереди
unsigned Uk2; // Указатель на «хвост» очереди
unsigned N; // Количество элементов очереди
};
void InitFifo(Fifo* f, unsigned SizeEl, unsigned SizeBuf);
// Инициализация очереди
void PutFifo(Fifo *f, void *E); /* Поместить элемент в очередь */
void GetFifo(Fifo *f, void *E); // Извлечь элемент из очереди
void ReadFifo(Fifo *f, void *E); // Прочитать элемент
int EmptyFifo(Fifo *f); // Проверка, пуста ли очередь?
void DoneFifo(Fifo *f); // Разрушить очередь
#endif
```

Задача 2: Система состоит из процессора P, трех очередей F_0, F_1, F_2 и стека S. В систему поступают запросы. Запрос можно представить записью.

```
typedef struct TInquiry
{
char Name[10]; // имя запроса
unsigned Time; // время обслуживания
char P; /* приоритет задачи: 0 — высший, 1 — средний, 2 — низший */
};
```

Поступающие запросы ставятся в соответствующие приоритетам очереди. Сначала обрабатываются задачи из очереди F_0. Если она пуста, можно обрабатывать задачи из очереди F_1. Если и она пуста, то можно обрабатывать задачи из очереди F_2. Если все очереди пусты, то система находится в ожидании поступающих задач (процессор свободен), либо в режиме

обработки предыдущей задачи (процессор занят). Если обрабатывается задача с низшим приоритетом и поступает задача с более высоким приоритетом, то обрабатываемая помещается в стек и может обрабатываться тогда и только тогда, когда все задачи с более высоким приоритетом уже обработаны.



Задание 2:

Начнем с реализации модуля для работы со стеком. Стек должен быть реализован на последовательном линейном списке, однако в прошлой лабораторной работе мы реализовали модуль для работы с односвязными линейными спиками. Значит, перед тем, как приступать к работе со стеками, необходимо написать модуль для ПЛС.

Как обычно, разделим код на два файла: заголовочный и реализации. В заголовочном файле зададим константы с кодами трех основных ошибок, которые могут возникнуть в ходе работы со списками: ListNotMem — для выделения места под новый элемент нет места в списке свободных элементов в массиве MemList; ListUnder — попытка получить доступ к элементу, в то время как пуст либо список, либо текущий указатель; ListEnd — в ходе перебора элементов списка был достигнут его конец. Под хранение кода ошибки отводится переменная. После этого дадим название используемым типам данных: ptrel — беззнаковое целое число, номер элемента списка, к которому идет обращение на данный момент; List — структура, хранящая массив с элементами типа void* (его размер задан константой), номер текущего элемента, их общее количество и размер базового типа. Только после этого будут объявлены прототипы функций для работы со списками.

```
1 #ifndef __LIST6_H
2 #define __LIST6_H
```

```
3
4  #define SizeList 100
5  const short ListOk = 0;
6  const short ListNotMem = 1;
7  const short ListUnder = 2;
8  const short ListEnd  = 3;
9  short ListError;
10
11 typedef void *BaseType;
12 typedef unsigned ptrel;
13 typedef struct {
14     BaseType MemList[SizeList];
15     ptrel ptr;
16     unsigned int N; // длина списка
17     unsigned int Size;
18 } List;
19
20 //Инициализация списка, который создается по адресу L
21 void InitList(List *L, unsigned Size);
22 //Включение элемента со значением E в список по адресу L
23 void PutList(List *L, BaseType E);
24 //Исключение элемента из списка по адресу L и сохранение его в
      переменной по адресу E
25 void GetList(List *L, BaseType *E);
26 //Чтение элемента списка по адресу L и сохранение его в переменной по
      адресу E
27 void ReadList(List *L,BaseType *E);
28 //Возвращает 1, если список по адресу L не пуст, и 0 в противном случае
29 int FullList(List *L);
30 //Проверка: является ли элемент списка по адресу L последним
31 int EndList(List *L);
32 //Возвращает количество элементов в списке по адресу L
33 unsigned int Count(List *L);
34 //Установка указателя в списке по адресу L в начало
35 void BeginPtr(List *L);
36 //Установка указателя в списке по адресу L в конец
37 void EndPtr(List *L);
38 //Переход к следующему элементу в списке по адресу L
39 void MovePtr(List *L);
40 //Переход к nму- элементу в списке по адресу L
41 void MoveTo(List *L, unsigned int n);
42 //Удаление списка по адресу L
43 void DoneList(List *L);
44 //Копирование списка по адресу L1 в список по адресу L2
45 void CopyList(List *L1, List *L2);
46 #endif
```

../АСД 6 си/SLList/SLList.h

Реализация функций будет вынесена в отдельный файл со следующим содержанием.

Примечание: функция PutList подразумевает, что элемент вставляется после элемента, номер которого хранит рабочий указатель, поэтому, чтобы вставить элемент в самое начало списка, рабочий указатель должен быть равен нулю, при условии, что нулю не равен стартовый указатель. Функция GetList подразумевает, что удаляется тот элемент, номер которого хранит рабочий указатель, поэтому попытка удаления элемента, когда указатель равен 0, приведет к выходу с кодом ошибки ListEnd. Нумерация начинается с единицы, при этом в массиве элементы хранятся, начиная с нулевого элемента: данное смещение учитывается в коде функций.

```
#ifndef __LIST6_C
#define __LIST6_C
#include "SLList.h"
#include <malloc.h>
#include <memory.h>

void InitList(List *L, unsigned Size){
    for (int i = 0; i < SizeList; i++) {
        L->MemList[i] = malloc(Size);
    }
    L->ptr = 0;
    L->N = 0;
    L->Size = Size;
}

void PutList(List *L, BaseType E){
    if (L->N == L->Size) {
        ListError = ListNotMem;
        exit(ListError);
    } else if (L->ptr > L->N) {
        ListError = ListEnd;
        exit(ListError);
    } else {
        for (ptrel cur_el = L->N; cur_el > L->ptr; cur_el--) {
            memcpy(L->MemList[cur_el], L->MemList[cur_el - 1],
   L->Size);
        }
        memcpy(L->MemList[L->ptr], E, L->Size);
        L->ptr += 1;
        L->N += 1;
        ListError = ListOk;
    }
}

void GetList(List *L, BaseType *E) {
    if (L->N == 0) {
        ListError = ListUnder;
```

```
        exit(ListError);
    } else if (L->ptr > L->N || L->ptr == 0) {
        ListError = ListEnd;
        exit(ListError);
    } else {
        memcpy(E, L->MemList[L->ptr-1], L->Size);
        for (ptrel cur_el = L->ptr-1; cur_el < L->N - 1; cur_el++) {
            L->MemList[cur_el] = L->MemList[cur_el + 1];
        }
        L->ptr--;
        L->N -= 1;
        ListError = ListOk;
    }
}

void ReadList(List *L, BaseType *E) {
    if (L->N == 0) {
        ListError = ListUnder;
        exit(ListError);
    } else if (L->ptr > L->N || L->ptr == 0) {
        ListError = ListEnd;
        exit(ListError);
    } else {
        memcpy(E, L->MemList[L->ptr-1], L->Size);
        ListError = ListOk;
    }
}

int FullList(List *L) {
    return L->N != 0;
}

int EndList(List *L) {
    return L->ptr == L->N;
}

unsigned int Count(List *L) {
    return L->N;
}

void BeginPtr(List *L) {
    if (L->N == 0) {
        L->ptr = 0;
    } else {
        L->ptr = 1;
    }
}
```

```
84
85 void EndPtr(List *L) {
86     L->ptr = L->N;
87 }
88
89 void MovePtr(List *L) {
90     L->ptr += 1;
91 }
92
93 void MoveTo(List *L, unsigned int n) {
94     L->ptr = n;
95 }
96
97 void DoneList(List *L) {
98     L->N = 0;
99     L->Size = 0;
100    L->ptr = 0;
101 }
102
103 void CopyList(List *L1, List *L2) {
104     for (ptrel cur_el = 0; cur_el < L1->N; cur_el++) {
105         memcpy(L2->MemList[cur_el], L1->MemList[cur_el], L1->Size);
106     }
107     L2->N = L1->N;
108     if (L2->ptr > L2->N) {
109         L2->ptr =  L2->N - 1;
110     }
111     ListError = ListOk;
112 }
113
114 #endif
```

../АСД 6 си/SLList/SLList.c

Теперь можем перейти к стеку. Также разобьем этот модуль на два файла. В предложенной реализации, структура «стек» — переименование структуры «список», и множество ошибок для нее — подмножество множества ошибок для структуры «список» (но эти множества не одинаковы; в отличие от списка, для стека не имеет смысл ошибка «достигнут конец»). Функции для стека тоже в каком-то смысле подмножеством функций списка.

```
1 #ifndef __STACK10_H
2 #define __STACK10_H
3 #include "../SLList/SLList.h"
4 const short StackOk = ListOk;
5 const short StackUnder = ListUnder;
6 const short StackOver = ListNotMem;
7 short StackError; // Переменная ошибок
8
9 typedef List Stack;
```

```
10 //Инициализация стека с максимлаьным размером size по адресу s
11 void InitStack(Stack *s, unsigned Size);
12 //Поместить элемент из переменной по адресу E в стек по адресу s
13 void PutStack(Stack *s, void *E);
14 //Извлечь элемент из стека по адресу s в переменную по адресу E
15 void GetStack(Stack *s, void *E);
16 //Проверка: пуст и стек по адресу s
17 int EmptyStack(Stack s);
18 //Прочитать элемент из вершины стека, записать в переменную по адресу E
19 void ReadStack(Stack s, void *E);
20 //Уничтожить стек по адресу s
21 void DoneStack(Stack *s);
22
23 #endif
```

../АСД 6 си/stack/stack.h

Также и реализация этих функций сводится к вызову аналогичных функций для списка, на который отображен стек, с единственной модификацией — указатель после выполнения каждой из них должен иметь значение 1 (указывать на первый элемент последовательности), поскольку вершиной стека считается первый элемент списка. При инициализации стека указатель приравнивается к 1: далее мы считаем, что он равен 1 по умолчанию, и устанавливаем его в 1 после вызова функций, которые могут его изменить. Единственный случай, когда мы устанавливаем его в 0 — включение элемента в стек (то есть вставка не после, а перед текущим первым элементом списка, которая требует именно такого значения указателя). А так как при работе со списками после включения элемента значение указателя возрастает на 1, нет нужды перезаписывать его как 1 повторно.

```
1  #ifndef __STACK10_C
2  #define __STACK10_C
3  #include "stack.h"
4  #include "../SLList/SLList.c"
5
6  void InitStack(Stack *s, unsigned Size) {
7      InitList(s, Size);
8      s->ptr = 1;
9  }
10
11 void PutStack(Stack *s, void *E) {
12     s->ptr = 0;
13     PutList(s, E);
14     s->ptr = 1;
15 }
16
17 void GetStack(Stack *s, void *E) {
18     GetList(s, E);
19     s->ptr = 1;
20 }
21
```

```
22 int EmptyStack(Stack s) {
23     return !FullList(&s);
24 }
25
26 void ReadStack(Stack s, void *E) {
27     ReadList(&s, E);
28     s.ptr = 1;
29 }
30
31 void DoneStack(Stack *s) {
32     DoneList(s);
33 }
34
35 #endif
```

../АСД 6 си/stack/stack.c

Для нормальных (не вызывающих ошибки и аварийного завершения) сценариев большинства функций (за исключением уничтожающей стек функции DoneStack), можно составить автоматизированные тесты и вынести их в отдельный файл тестирования. Будем тестировать функции по порядку; после того, как они протестированы, их можно использовать в тестах дальнейших функций. Запустив программу, можем самостоятельно убедиться, что все тесты прошли успешно:

```
1  #include "stack.c"
2  #include <assert.h>
3  #include <stdio.h>
4
5
6  void Test_InitStack() {
7      Stack S;
8      InitStack(&S, sizeof(int));
9      assert(S.ptr == 1 && S.N == 0 && S.Size == sizeof(int));
10     DoneStack(&S);
11 }
12
13 void Test_PutStack_ToEmpty() {
14     Stack S;
15     InitStack(&S, sizeof(int));
16     int a = 3;
17     PutStack(&S, &a);
18     assert(memcmp(&a, S.MemList[0], sizeof(int)) == 0 && S.ptr == 1 &&
       S.N == 1);
19     DoneStack(&S);
20 }
21 void Test_PutStack_ToNotEmpty() {
22     Stack S;
23     InitStack(&S, sizeof(int));
24     int a = 3;
```

```
25      int b = 5;
26      PutStack(&S, &a);
27      PutStack(&S, &b);
28      assert(memcmp(&a, S.MemList[1], sizeof(int)) == 0);
29      assert(memcmp(&b, S.MemList[0], sizeof(int)) == 0 && S.ptr == 1 &&
        S.N == 2);
30      DoneStack(&S);
31  }
32  void Test_PutStack() {
33      Test_PutStack_ToEmpty();
34      Test_PutStack_ToNotEmpty();
35  }
36
37  void Test_GetStack_One() {
38      Stack S;
39      InitStack(&S, sizeof(int));
40      int a = 3;
41      PutStack(&S, &a);
42      int dest_a;
43      GetStack(&S, &dest_a);
44      assert(dest_a == 3 && S.ptr == 1 && S.N == 0);
45      DoneStack(&S);
46  }
47  void Test_GetStack_NotOne() {
48      Stack S;
49      InitStack(&S, sizeof(int));
50      int a = 3;
51      int b = 5;
52      PutStack(&S, &a);
53      PutStack(&S, &b);
54      int dest_b, dest_a;
55      GetStack(&S, &dest_b);
56      GetStack(&S, &dest_a);
57      assert(dest_b == 5);
58      assert(dest_a == 3 && S.ptr == 1 && S.N == 0);
59      DoneStack(&S);
60  }
61  void Test_GetStack() {
62      Test_GetStack_One();
63      Test_GetStack_NotOne();
64  }
65
66  void Test_EmptyStack_Empty() {
67      Stack S;
68      InitStack(&S, sizeof(int));
69      assert(EmptyStack(S));
70      DoneStack(&S);
```

```
}
void Test_EmptyStack_One() {
    Stack S;
    InitStack(&S, sizeof(int));
    int a = 3;
    PutStack(&S, &a);
    assert(!EmptyStack(S));
    DoneStack(&S);
}
void Test_EmptyStack_NotOne() {
    Stack S;
    InitStack(&S, sizeof(int));
    int a = 3;
    int b = 5;
    PutStack(&S, &a);
    PutStack(&S, &b);
    assert(!EmptyStack(S));
    DoneStack(&S);
}
void Test_EmptyStack() {
    Test_EmptyStack_Empty();
    Test_EmptyStack_One();
    Test_EmptyStack_NotOne();
}

void Test_ReadStack_One() {
    Stack S;
    InitStack(&S, sizeof(int));
    int a = 3;
    PutStack(&S, &a);
    int dest_a;
    ReadStack(S, &dest_a);
    assert(dest_a == 3 && S.ptr == 1 && S.N == 1);
    DoneStack(&S);
}
void Test_ReadStack_NotOne() {
    Stack S;
    InitStack(&S, sizeof(int));
    int a = 3;
    int b = 5;
    PutStack(&S, &a);
    PutStack(&S, &b);
    int dest_b, dest_b2;
    ReadStack(S, &dest_b);
    ReadStack(S, &dest_b2);
    assert(dest_b == 5);
    assert(dest_b2 == 5 && S.ptr == 1 && S.N == 2);
```

```
118     DoneStack(&S);
119 }
120 void Test_ReadStack() {
121     Test_ReadStack_One();
122     Test_ReadStack_NotOne();
123 }
124 
125 void Test_Stack () {
126     Test_InitStack();
127     Test_PutStack();
128     Test_GetStack();
129     Test_EmptyStack();
130     Test_ReadStack();
131 }
132 
133 int main() {
134     Test_Stack();
135     printf("All is OK!");
136 }
```

../АСД 6 си/stack/stack_test.c

```
"C:\Users\sovac\Desktop\ASD_third_semester\АСД 6 си\stack\stack_test.exe"
All is OK!
Process finished with exit code 0
```

Перейдем к очереди. Кольцевая очередь реализуется непосредственно на массиве. Также определяется список ошибок: FifoUnder, если происходит попытка извлечь/прочесть элемент из пустой очереди, и FifoOver, если количество элементов в очереди превышает количество элементов, для которого изначально была отведена память. После этого задается структура «очередь» с шестью полями: указатель на начало массива, элементы которого — пустые указатели на реальные элементы очереди; максимальное количество элементов, для хранения которых выделяется память при инициализации, размер базового типа в байтах, индекс элемента в массиве, являющегося головой очереди, индекс элемента в массиве, являющегося хвостом очереди, переменная, хранящая количество элементов в массиве.

```
1  #if !defined(__FIFO11_H)
2  #define __FIFO11_H
3  const int FifoOk = 0;
4  const int FifoUnder = 1;
5  const int FifoOver = 2;
6  int FifoError; // Переменная ошибок
7  
8  typedef void *BaseType;
9  typedef struct {
10     BaseType *Buf;
11     unsigned SizeBuf; // Максимальная длина очереди
12     unsigned SizeEl; // Размер элемента очереди
13     unsigned Uk1; // Указатель на «голову» очереди
```

```
    unsigned Uk2; // Указатель на «хвост» очереди
    unsigned N;   // Количество элементов очереди
} Fifo;

// Инициализация очереди по адресу f
void InitFifo(Fifo* f, unsigned SizeEl, unsigned SizeBuf);
//Поместить элемент из переменной по адресу E в очередь по адресу f
void PutFifo(Fifo *f, void *E);
//Извлечь элемент из очереди по адресу f и записать его в переменую по
   адресу E
void GetFifo(Fifo *f, void *E);
//Прочитать элемент очереди по адресу f и записать его в переменую по
   адресу E
void ReadFifo(Fifo *f, void *E);
//Проверка, пуста ли очередь по адресу f
int EmptyFifo(Fifo *f);
//Разрушить очередь по адресу f
void DoneFifo(Fifo *f);

#endif
```

../АСД 6 си/fifo/fifo.h

Работая со стеком на последовательном линейном списке, мы условились, что его врешиной будет первый элемент списка, и, включая или исключая элемент, сдвигали уже имеющиеся относительно начала. В данной реализации очереди мы поступим противоположным образом: элементы будут оставаться на прежних позициях, но указатели на голову и хвост будут смещаться таким образом, что указатель на голову всегда указывает на последний (головной) элемент очереди, указатель на хвост — на позицию, следующую за первым (хвостовым) элементом. При операции включения указатель на хвост увеличивается на 1, при операции исключения увеличивается на 1 указатель на голову. Таким образом, област ьмежду указателями, в которой и будут храниться элементы, будет постепенно смещаться от его левого края к правому, оставляя «по бокам» неиспользованное место. Но очередь, которую мы реализуем, является кольцевой; достигая конца массива, указатель переходит на его начало. Таким образом, очередь может использовать область массива «по бокам», а последовательность элементов может быть распределена так, что часть ее продолжается от головы до правого конца массива, а вторая часть — от начала массива до хвоста. Это учитывается в теле функций включения и исключения.

```
#if !defined(__FIFO11_C)
#define __FIFO11_C
#include "fifo.h"
#include <malloc.h>
#include <memory.h>
#include <stdlib.h>

void InitFifo(Fifo* f, unsigned SizeEl, unsigned SizeBuf) {
    f->Buf = malloc( SizeBuf * sizeof(void*));
    for (int i = 0; i < SizeBuf; i++) {
        f->Buf[i] = malloc(SizeEl);
```

```
    }
    f->SizeEl = SizeEl;
    f->SizeBuf = SizeBuf;
    f->Uk1 = 0;
    f->Uk2 = 0;
    f->N = 0;
}

void PutFifo(Fifo *f, void *E) {
    if (f->N == f->SizeBuf) {
        FifoError = FifoOver;
        exit(FifoError);
    } else {
        memcpy(f->Buf[f->Uk2], E, f->SizeEl);
        f->Uk2 = (f->Uk2 + 1) % f->SizeBuf;
        f->N += 1;
        FifoError = FifoOk;
    }
}

void GetFifo(Fifo *f, void *E) {
    if (f->N == 0){
        FifoError = FifoUnder;
        exit(FifoUnder);
    } else {
        memcpy(E, f->Buf[f->Uk1], f->SizeEl);
        f->Uk1 = (f->Uk1 + 1) % f->SizeBuf;
        f->N -= 1;
        FifoError = FifoOk;
    }
}

void ReadFifo(Fifo *f, void *E) {
    if (f->N == 0){
        FifoError = FifoUnder;
        exit(FifoUnder);
    } else {
        memcpy(E, f->Buf[f->Uk1], f->SizeEl);
        FifoError = FifoOk;
    }
}

int EmptyFifo(Fifo *f) {
    return f->N == 0;
}

void DoneFifo(Fifo *f) {
```

```
    for (int i = 0; i < f->SizeBuf; i++) {
        free((f->Buf)[i]);
    }
    free(f->Buf);
    f->SizeBuf = 0;
    f->SizeEl = 0;
    f->Uk1 = 0;
    f->Uk2 = 0;
    f->N = 0;
}


#endif
```

../АСД 6 си/fifo/fifo.c

Аналогичным образом проведем тестирование для очереди (нормальных сценариев всех функций, кроме DoneFifo). Эти тесты также прошли успешно:

```
#include "fifo.c"
#include <assert.h>
#include <stdio.h>


void Test_InitFifo() {
    Fifo F;
    InitFifo(&F, sizeof(int), 5);
    assert(F.SizeEl == sizeof(int) && F.SizeBuf == 5 && F.N == 0 &&
   F.Uk1 == 0 && F.Uk2 == 0);
    DoneFifo(&F);
}

void Test_PutFifo_ToEmpty() {
    Fifo F;
    InitFifo(&F, sizeof(int), 5);
    int a = 3;
    PutFifo(&F, &a);
    assert(memcmp(&a, F.Buf[0], sizeof(int)) == 0 && F.Uk2 == 1 && F.N
   == 1);
    DoneFifo(&F);
}
void Test_PutFifo_ToNotEmpty() {
    Fifo F;
    InitFifo(&F, sizeof(int), 5);
    int a = 3;
    int b = 5;
    PutFifo(&F, &a);
    PutFifo(&F, &b);
    assert(memcmp(&a, F.Buf[0], sizeof(int)) == 0);
```

```
29      assert(memcmp(&b, F.Buf[1], sizeof(int)) == 0 && F.Uk2 == 2 && F.N
        == 2);
30      DoneFifo(&F);
31  }
32  void Test_PutFifo() {
33      Test_PutFifo_ToEmpty();
34      Test_PutFifo_ToNotEmpty();
35  }
36
37  void Test_GetFifo_One() {
38      Fifo F;
39      InitFifo(&F, sizeof(int), 5);
40      int a = 3;
41      PutFifo(&F, &a);
42      int dest_a;
43      GetFifo(&F, &dest_a);
44      assert(dest_a == 3 && F.Uk1 == 1 && F.N == 0);
45      DoneFifo(&F);
46  }
47  void Test_GetFifo_NotOne() {
48      Fifo F;
49      InitFifo(&F, sizeof(int), 5);
50      int a = 3;
51      int b = 5;
52      PutFifo(&F, &a);
53      PutFifo(&F, &b);
54      int dest_a, dest_b;
55      GetFifo(&F, &dest_a);
56      GetFifo(&F, &dest_b);
57      assert(dest_b == 5);
58      assert(dest_a == 3 && F.Uk1 == 2 && F.N == 0);
59      DoneFifo(&F);
60  }
61  void Test_GetFifo() {
62      Test_GetFifo_One();
63      Test_GetFifo_NotOne();
64  }
65
66  void Test_EmptyFifo_Empty() {
67      Fifo F;
68      InitFifo(&F, sizeof(int), 5);
69      assert(EmptyFifo(&F));
70      DoneFifo(&F);
71  }
72  void Test_EmptyFifo_One() {
73      Fifo F;
74      InitFifo(&F, sizeof(int), 5);
```

```
75      int a = 3;
76      PutFifo(&F, &a);
77      assert(!EmptyFifo(&F));
78      DoneFifo(&F);
79  }
80  void Test_EmptyFifo_NotOne() {
81      Fifo F;
82      InitFifo(&F, sizeof(int), 5);
83      int a = 3;
84      int b = 5;
85      PutFifo(&F, &a);
86      PutFifo(&F, &b);
87      assert(!EmptyFifo(&F));
88      DoneFifo(&F);
89  }
90  void Test_EmptyFifo() {
91      Test_EmptyFifo_Empty();
92      Test_EmptyFifo_One();
93      Test_EmptyFifo_NotOne();
94  }
95
96  void Test_ReadFifo_One() {
97      Fifo F;
98      InitFifo(&F, sizeof(int), 5);
99      int a = 3;
100     PutFifo(&F, &a);
101     int dest_a;
102     ReadFifo(&F, &dest_a);
103     assert(dest_a == 3 && F.Uk1 == 0 && F.N == 1);
104     DoneFifo(&F);
105 }
106 void Test_ReadFifo_NotOne() {
107     Fifo F;
108     InitFifo(&F, sizeof(int), 5);
109     int a = 3;
110     int b = 5;
111     PutFifo(&F, &a);
112     PutFifo(&F, &b);
113     int dest_a, dest_a2;
114     ReadFifo(&F, &dest_a);
115     ReadFifo(&F, &dest_a2);
116     assert(dest_a == 3);
117     assert(dest_a2 == 3 && F.Uk1 == 0 && F.N == 2);
118     DoneFifo(&F);
119 }
120 void Test_ReadFifo() {
121     Test_ReadFifo_One();
```

```
122     Test_ReadFifo_NotOne();
123 }
124
125
126 void Test_Fifo () {
127     Test_InitFifo();
128     Test_PutFifo();
129     Test_GetFifo();
130     Test_EmptyFifo();
131     Test_ReadFifo();
132 }
133
134 int main() {
135     Test_Fifo();
136     printf("All is OK!");
137 }
```

../АСД 6 си/fifo/fifo_test.c

```
"C:\Users\sovac\Desktop\ASD_third_semester\АСД 6 си\fifo\fifo_test.exe"
All is OK!
Process finished with exit code 0
```

Задание 3:

Теперь, когда модули для СД типа «стек» и «очередь», реализованы, можем перейти к решению задачи: моделированию вычислительной системы, описанной выше. Сначала объявим структуры для отображения объектов, использующихся в модели системы, помимо стека и очередей — процессора и генератора задач. Затем объявим функции, отвечающие за некоторые ключевые шаги (к каждой из них написан комментарий, объясняющий, за какое действие в системе она отвечает), и функции для вывода задач, содержащихся в той или иной структуре. Все эти функции будут объединены в функции makeTact, которая будет выполнять вышеописанные шаги в зависимости от текущего наполнения очередей, стека, процессора, и выводить их на экран в конце такта. В теле функции main все составляющие модели инициализируются, и функция makeTact запускается в цикле заданное количество раз.

```
1 #include <stdbool.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include "../stack/stack.c"
5 #include "../fifo/fifo.c"
6
7 typedef struct {
8     char Name[10]; //имя запроса
9     unsigned Time; //время обслуживания
```

```
10      char P;         //приоритет задачи: 0 – высший, 1 – средний, 2 –
        низший
11  } TInquiry;
12
13  typedef struct {
14      TInquiry *cur_task;     //указатель на содержащуюся в нем задачу
15      bool has_task;          //флаг: находится ли задача по указателю
16      unsigned tasks_before;  //количество уже сгенерированных задач
17  } Generator;
18
19  typedef struct {
20      TInquiry *cur_task; //указатель на содержащуюся в нем задачу
21      bool has_task;      //флаг: находится ли задача по указателю
22      char delayedP;      //Приоритет отложенной задачи; если отложенной
        задачи нет, равен 3
23  } Processor;
24
25  //Возвращает значение истина"" с вероятностью 60 процентов
26  bool shouldGenerateTask() {
27      return rand() % 100 < 60;
28  }
29
30  //Задает приоритет со следующим распределением: 30% - приоритет 0, 30%
        - приоритет 1, 40% - приоритет 2
31  int generatePriority() {
32      int num = rand() % 100;
33      if (num < 30) {
34          return 0;
35      } else if (num < 60) {
36          return 1;
37      } else {
38          return 2;
39      }
40  }
41
42  //Записывает по адресу cur_task генератора задач G задачу с:
43  // 1) именем из случайных символов их( количество может варьироваться
        от 2 до 7) и порядкового кномера задачи,
44  // 2) случайным временем выполнения варьируется( от 2 до 7 тактов),
45  // 3) случайным приоритетом, определяемым значением функции
        generatePriority
46  //После этого к счетчику сгенерированных задач прибавляется 1, а флаг,
        обозначающий, находится ли в массиве задача, принимает значение
        истина""
47  void generateTInquiry(Generator *G) {
48      char *alphabet = "abcdefghijklmnopqrstuvwxyz";
49      int name_len = 2 + rand() % 6;
```

```
50      for (int i = 0; i < name_len; i++) {
51          G->cur_task->Name[i] = alphabet[rand()%26];
52      }
53      snprintf(G->cur_task->Name + name_len, 3, "%d", G->tasks_before);
54      G->cur_task->Time = 2 + rand()%6;
55      G->cur_task->P = generatePriority();
56      G->has_task = true;
57      G->tasks_before += 1;
58  }
59
60  //Если генератор задач G содержит задачу, переносит ее в одлну из
        очередей F1, F2, F3 в зависимости от ее приоритета
61  //После этого флаг, обозначающий, находится ли в массиве задача,
        принимает значение ложь""
62  void transferTaskToFifo(Generator *G, Fifo *F1, Fifo *F2, Fifo *F3) {
63      if (G->has_task) {
64          if (G->cur_task->P == 0) {
65              PutFifo(F1, G->cur_task);
66          } else if (G->cur_task->P == 1) {
67              PutFifo(F2, G->cur_task);
68          } else {
69              PutFifo(F3, G->cur_task);
70          }
71          G->has_task = 0;
72      }
73  }
74
75  //Возвращает наивысший приоритет среди содержащихся в очереди задач
76  //Приоритеты 0, 1, 2 соотвествуют очередям F1, F2, F3 соответственно.
        Если все эти очереди пусты, возвращается 3
77  char findHighestPriority(Fifo *F1, Fifo *F2, Fifo *F3) {
78      if (!EmptyFifo(F1)) {
79          return 0;
80      } else if (!EmptyFifo(F2)) {
81          return 1;
82      } else if (!EmptyFifo(F3)) {
83          return 2;
84      } else {
85          return 3;
86      }
87  }
88
89  //Переносит из непустой очереди с наивысшим приоритетом среди(
        очередей F1, F2, F3) задачу в процессор P, устанавливает флаг
        наличия задачи в процессоре в истина""
90  //Если непустая очередь не найдена, переноса не происходит
```

```
91  void transferTaskToProcessor(Processor *P, Fifo *F1, Fifo *F2, Fifo
      *F3) {
92      char priority = findHighestPriority(F1, F2, F3);
93      if (priority == 0) {
94          GetFifo(F1, P->cur_task);
95          P->has_task = 1;
96      } else if (priority == 1) {
97          GetFifo(F2, P->cur_task);
98          P->has_task = 1;
99      } else if (priority == 2) {
100         GetFifo(F3, P->cur_task);
101         P->has_task = 1;
102     }
103 }
104
105 //Выполняет перенос задачи из процессора P в стек S, после чего
      заполняет процессор новой задачей из очереди с наивысшим приоритетом
106 void shiftOnHigherPriority(Processor *P, Stack *S, Fifo *F1, Fifo *F2,
      Fifo *F3) {
107     P->delayedP = P->cur_task->P;
108     PutStack(S, P->cur_task);
109     transferTaskToProcessor(P, F1, F2, F3);
110 }
111
112 //Считывает последнюю задачу в стеке отложенных задач S и на ее
      основании обновляет параметр delayedP в процессорве P
113 void updateDelayedP(Processor *P, Stack *S) {
114     if (!EmptyStack(*S)) {
115         TInquiry last_delayed;
116         ReadStack(*S, &last_delayed);
117         P->delayedP = last_delayed.P;
118     } else {
119         P->delayedP = 3;
120     }
121 }
122
123 //Выводит на экран задачу T в формате имявремяприоритет(;;)
124 void outputTInquiry(TInquiry *T) {
125     printf("(%s;%d)", T->Name, T->Time);
126 }
127
128 //Выводит на экран задачу, содержащуюся в процессоре P; если процессор
      не содержит задачу, выводится сообщение "empty"
129 void outputProcessor (Processor *P) {
130     if (P->has_task == 0) {
131         printf("empty");
132     } else {
```

```
133         outputTInquiry(P->cur_task);
134     }
135 }
136 
137 //Выводит на экран задачи, содержащиеся в очереди F; если очередь не
      содержит задач, выводится сообщение "empty"
138 void outputFifo (Fifo *F) {
139     if (EmptyFifo(F)) {
140         printf("empty");
141     } else {
142         Fifo tempFifo;
143         InitFifo(&tempFifo, F->SizeEl, F->SizeBuf);
144         TInquiry *task = malloc(sizeof(TInquiry));
145         while (!EmptyFifo(F)) {
146             GetFifo(F, task);
147             outputTInquiry(task);
148             printf(",");
149             PutFifo(&tempFifo, task);
150         }
151         printf("\b ");
152         while (!EmptyFifo(&tempFifo)) {
153             GetFifo(&tempFifo, task);
154             PutFifo(F, task);
155         }
156         DoneFifo(&tempFifo);
157     }
158 }
159 
160 //Выводит на экран задачи, содержащиеся в стеке S; если стек не
      содержит задач, выводится сообщение "empty"
161 void outputStack (Stack *S) {
162     if (EmptyStack(*S)) {
163         printf("empty");
164     } else {
165         Stack tempStack;
166         InitStack(&tempStack, S->Size);
167         TInquiry *task = malloc(sizeof(TInquiry));
168         while (!EmptyStack(*S)) {
169             GetStack(S, task);
170             outputTInquiry(task);
171             printf(",");
172             PutStack(&tempStack, task);
173         }
174         printf("\b ");
175         while (!EmptyStack(tempStack)) {
176             GetStack(&tempStack, task);
177             PutStack(S, task);
```

```
178         }
179         DoneStack(&tempStack);
180     }
181 }
182 
183 //Моделирует выполнение одного такта системы из генератора задач G,
    процессора P, стека S и очередей F1, F2, F3
184 //Выводит состояние всех компонентов системы, кроме генератора задач,
    по завершению такта
185 void makeTact(Generator *G, Processor *P, Stack *S, Fifo *F1, Fifo
    *F2, Fifo *F3) {
186     if (shouldGenerateTask() && G->has_task == 0) {
187         generateTInquiry(G);
188     }
189     transferTaskToFifo(G, F1, F2, F3);
190     if (P->has_task == 0) {
191         if (!EmptyStack(*S) && P->delayedP <= findHighestPriority(F1,
    F2, F3)) {
192             GetStack(S, P->cur_task);
193             P->has_task = 1;
194             updateDelayedP(P, S);
195         } else {
196             transferTaskToProcessor(P, F1, F2, F3);
197         }
198     } else if (findHighestPriority(F1, F2, F3) < P->cur_task->P) {
199         shiftOnHigherPriority(P, S, F1, F2, F3);
200     }
201     if (P->has_task) {
202         P->cur_task->Time -= 1;
203         if (P->cur_task->Time == 0) {
204             P->has_task = 0;
205         }
206     }
207     printf("Fifo No1: ");
208     outputFifo(F1);
209     printf("\nFifo No2: ");
210     outputFifo(F2);
211     printf("\nFifo No3: ");
212     outputFifo(F3);
213     printf("\nStack: ");
214     outputStack(S);
215     printf("\nProcessor: ");
216     outputProcessor(P);
217 }
218 
219 int main() {
220     Generator G;
```

```
221     Processor P;
222     Fifo F1, F2, F3;
223     Stack S;
224     InitFifo(&F1, sizeof(TInquiry), 50);
225     InitFifo(&F2, sizeof(TInquiry), 50);
226     InitFifo(&F3, sizeof(TInquiry), 50);
227     InitStack(&S, sizeof(TInquiry));
228     G.has_task = 0;
229     G.tasks_before = 0;
230     P.has_task = 0;
231     P.delayedP = 3;
232 
233     for (int time = 1; time <= 20; time++) {
234         printf("\n\nTIME: %d\n", time);
235         makeTact(&G, &P, &S, &F1, &F2, &F3);
236     }
237 }
```

../АСД 6 си/task/task_updated.c

Пример, показывающий, как организован вывод, можно увидеть на скриншотах ниже. На них запечатлен вывод после выполнения десяти тактов:

```
TIME: 1
Fifo No1: empty
Fifo No2: empty
Fifo No3: empty
Stack: empty
Processor: (qghumea0;3)

TIME: 2
Fifo No1: empty
Fifo No2: (fdx1;3)
Fifo No3: empty
Stack: empty
Processor: (qghumea0;2)

TIME: 3
Fifo No1: empty
Fifo No2: (fdx1;3),(vs2;4)
Fifo No3: empty
Stack: empty
Processor: (qghumea0;1)

TIME: 4
Fifo No1: empty
Fifo No2: (fdx1;3),(vs2;4)
Fifo No3: empty
Stack: empty
Processor: empty

TIME: 5
Fifo No1: empty
Fifo No2: (vs2;4)
Fifo No3: empty
Stack: empty
Processor: (fdx1;2)
```

```
TIME: 6
Fifo No1: empty
Fifo No2: (vs2;4),(kfnq3;7)
Fifo No3: empty
Stack: empty
Processor: (fdx1;1)

TIME: 7
Fifo No1: empty
Fifo No2: (vs2;4),(kfnq3;7)
Fifo No3: empty
Stack: empty
Processor: empty

TIME: 8
Fifo No1: empty
Fifo No2: (kfnq3;7)
Fifo No3: (nfozvsr4;5)
Stack: empty
Processor: (vs2;3)

TIME: 9
Fifo No1: empty
Fifo No2: (kfnq3;7)
Fifo No3: (nfozvsr4;5),(rep5;4)
Stack: empty
Processor: (vs2;2)

TIME: 10
Fifo No1: empty
Fifo No2: (kfnq3;7),(pnrvy6;6)
Fifo No3: (nfozvsr4;5),(rep5;4)
Stack: empty
Processor: (vs2;1)
```

Перенесем в таблицу выведенные данные.

Время	**Объекты**	**Задачи**
1	F1	empty
	F2	empty
	F3	empty
	S	empty
	P	(qghumea0;3)
2	F1	empty
	F2	(fdx1;3)
	F3	empty
	S	empty
	P	(qghumea0;2)
3	F1	empty
	F2	(fdx1;3),(vs2;4)
	F3	empty
	S	empty
	P	(qghumea0;1)
4	F1	empty
	F2	(fdx1;3),(vs2;4)
	F3	empty
	S	empty
	P	empty
5	F1	empty
	F2	(vs2;4)
	F3	empty
	S	empty
	P	(fdx1;2)
6	F1	empty
	F2	(vs2;4),(kfnq3;7)
	F3	empty
	S	empty
	P	(fdx1;1)
7	F1	empty
	F2	(vs2;4),(kfnq3;7)
	F3	empty
	S	empty
	P	empty
8	F1	empty
	F2	(kfnq3;7)
	F3	(nfozvsr4;5)
	S	empty
	P	(vs2;3)

9	F1	empty
	F2	(kfnq3;7)
	F3	(nfozvsr4;5),(rep5;4)
	S	empty
	P	(vs2;2)
10	F1	empty
	F2	(kfnq3;7),(pnrvy6;6)
	F3	(nfozvsr4;5),(rep5;4)
	S	empty
	P	(vs2;1)
11	F1	empty
	F2	(kfnq3;7),(pnrvy6;6)
	F3	(nfozvsr4;5),(rep5;4)
	S	empty
	P	empty
12	F1	empty
	F2	(pnrvy6;6)
	F3	(nfozvsr4;5),(rep5;4),(ys7;4)
	S	empty
	P	(kfnq3;6)
13	F1	empty
	F2	(pnrvy6;6)
	F3	(nfozvsr4;5),(rep5;4),(ys7;4)
	S	(kfnq3;6)
	P	(pevi8;3)
14	F1	empty
	F2	(pnrvy6;6),(mznim9;2)
	F3	(nfozvsr4;5),(rep5;4),(ys7;4)
	S	(kfnq3;6)
	P	(pevi8;2)
15	F1	empty
	F2	(pnrvy6;6),(mznim9;2)
	F3	(nfozvsr4;5),(rep5;4),(ys7;4)
	S	(kfnq3;6)
	P	(pevi8;1)
16	F1	empty
	F2	(pnrvy6;6),(mznim9;2)
	F3	(nfozvsr4;5),(rep5;4),(ys7;4)
	S	(kfnq3;6)
	P	empty
17	F1	empty
	F2	(pnrvy6;6),(mznim9;2)
	F3	(nfozvsr4;5),(rep5;4),(ys7;4),(srenzk10;4)
	S	empty
	P	(kfnq3;5)

18	F1	empty
	F2	(pnrvy6;6),(mznim9;2)
	F3	(nfozvsr4;5),(rep5;4),(ys7;4),(srenzk10;4),(xtlsgyp11;4)
	S	empty
	P	(kfnq3;4)
19	F1	empty
	F2	(pnrvy6;6),(mznim9;2)
	F3	(nfozvsr4;5),(rep5;4),(ys7;4),(srenzk10;4),(xtlsgyp11;4)
	S	empty
	P	(kfnq3;3)
20	F1	empty
	F2	(pnrvy6;6),(mznim9;2)
	F3	(nfozvsr4;5),(rep5;4),(ys7;4),(srenzk10;4),(xtlsgyp11;4),(ooefxzb12;2)
	S	empty
	P	(kfnq3;2)

Вывод:

В ходе лабораторной работы дали характеристику СД типа «стек» и «очередь», форматам их представления, реализовали по одному из них для каждой СД из них (стек на базе последовательного линейного списка с вершиной в первом элементе и кольцевую очередь на базе динамического массива), написали ряд базовых функций для работы со стеками и очередями в этом формате, а также создали модель вычислительной системы, используя, в том числе, СД типа «стек» и «очередь».