**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ**

**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

автоматизированныхсистем

Лабораторная работа №5

дисциплина: Алгоритмы и структуры данных

тема: «Структуры данных «стек» и «очередь» (Pascal/С)»

Выполнил ст. группы

Проверил: проф. Синюк В.Г.

Белгород 20

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

1.1. Абстрактный уровень представления СД:

* + 1. Характер организованности и изменчивости.

Для стека: линейная структура - последовательность.

Для очереди: линейная структура – последовательность.

* + 1. Набор допустимых операций.

Для стека: инициализация, включение, исключение, чтение, проверка пустоты, уничтожение.

Для очереди: инициализация, включение, исключение, проверка пустоты, уничтожение.

1.2. Физический уровень представления СД:

1.2.1. Схему хранения.

Для стека: последовательная или связная.

Для очереди: последовательная или связная.

1.2.2. Объем памяти, занимаемый экземпляром СД.

Для стека: зависит от максимального числа элементов в стеке

Для очереди: зависит от максимального числа элементов в очереди

1.2.3. Формат внутреннего представления СД и способ его интерпретации.

Для стека: может использоваться как статический и динамический массив, или реализованный на них список.

Для очереди: аналогично стеку.

1.2.4. Характеристику допустимых значений.

Для стека: CAR(стек) = CAR(BaseType)0 + CAR(BaseType)1 + … + CAR(BaseType)max,

Для очереди: CAR(FIFO) = CAR(BaseType)0 + CAR(BaseType)1 + … + CAR(BaseType)max

1.2.5. Тип доступа к элементам.

Для стека: в зависимости от СД, на котором реализован.

Для очереди: в зависимости от СД, на котором реализована.

1.3. Логический уровень представления СД.

Способ описания СД и экземпляра СД на языке программирования.

**Стек**

const short StackSize = 100;

typedef int BaseType;

typdef struct {

BaseType Buf [StackSize];

unsigned uk;

} Stack;

**Очередь**

const short FifoSize = 100;

typedef int BaseType;

typdef struct {

BaseType Buf [FifoSize];

unsigned uk1;

unsigned uk2;

unsigned n;

} Fifo;

**List.h**

#ifndef **ASD\_6\_LIST\_H**#define **ASD\_6\_LIST\_H  
static const short** ListOk = 0;  
**static const short** ListEmpty = 1;  
**static const short** ListNotMem = 2;  
**static const short** ListEnd = 4;  
  
**short** ListError;  
  
**typedef struct**{  
 **int** data;  
 **unsigned** time;  
 **int** p;  
} TInquiry;

**typedef** TInquiry BaseType;  
**typedef struct** element\* elptr;  
  
**typedef struct** element  
{  
 BaseType data;  
 elptr next;  
};  
  
**typedef struct**{  
 elptr start;  
 elptr ptr;  
} List;  
  
**void** InitList(List\* l);  
**void** PutList(List\* l, BaseType e);  
**void** GetList(List\* l, BaseType\* e);  
**short** EmptyList(List\* l);  
**void** BeginListPtr(List\* l);  
**void** EndListPtr(List\* l);  
**void** MovePtr(List\* l);  
**void** DoneList(List\* l);  
  
#endif *//ASD\_6\_LIST\_H*

**List.c**

#include **<stdlib.h>**#include **"List.h"  
  
void** InitList(List\* l)  
{  
 elptr pntr = (elptr) malloc(**sizeof**(**struct** element));  
  
 **if**(pntr != **NULL**)  
 {  
 pntr->next = **NULL**;  
 }  
 **else** {  
 ListError = ListNotMem;  
 **return**;  
 }  
  
 l->start = pntr;  
 l->ptr = pntr;  
 ListError = ListOk;  
}  
  
**void** PutList(List\* l, BaseType e)  
{  
 elptr pntr = (elptr)malloc(**sizeof**(**struct** element));  
  
 **if** (pntr != **NULL**)  
 {  
 pntr->data = e;  
 pntr->next = **NULL**;  
 }  
 **else** {  
 ListError = ListNotMem;  
 **return**;  
 }  
  
 pntr->next = l->ptr->next;  
 l->ptr->next = pntr;  
 ListError = ListOk;  
}  
  
**void** GetList(List\* l, BaseType\* e)  
{  
 **if** (l->start != l->ptr && l->ptr->next == **NULL**)  
 {  
  
 ListError = ListEnd;  
 }  
 **else if** (EmptyList(l))  
 {  
 ListError = ListEmpty;  
 }  
 **else** {  
 elptr pntr = l->ptr->next;  
 \*e = pntr->data;  
 l->ptr->next = pntr->next;  
 free(pntr);  
 ListError = ListOk;  
 }  
}  
  
**short** EmptyList(List\* l)  
{  
 **return** l->start->next == **NULL**;  
}  
  
**void** BeginListPtr(List\* l)  
{  
 **if** (EmptyList(l))  
 ListError = ListEmpty;  
 **else** {  
 l->ptr = l->start;  
 ListError = ListOk;  
 }  
}  
  
**void** EndListPtr(List\* l)  
{  
 **if** (EmptyList(l))  
 ListError = ListEmpty;  
 **else** {  
 BeginListPtr(l);  
  
 **while** (l->ptr->next != **NULL**)  
 MovePtr(l);  
 }  
}  
  
**void** MovePtr(List\* l)  
{  
 **if** (EmptyList(l))  
 ListError = ListEmpty;  
 **else if** (l->ptr->next == **NULL**)  
 ListError = ListEnd;  
 **else** {  
 l->ptr = l->ptr->next;  
 ListError = ListOk;  
 }  
}  
  
**void** DoneList(List\* l)  
{  
 BeginListPtr(l);  
  
 **while** (!EmptyList(l))  
 {  
 BaseType null;  
  
 GetList(l, &null);  
 }  
  
 free(l->start);  
}

**Stack.h**

#ifndef **ASD\_6\_STACK\_H**#define **ASD\_6\_STACK\_H**#include **"List.h"  
  
static const short** StackOk = 0; *// ListOk***static const short** StackEmpty = 1; *// ListEmpty***static const short** StackNotMem = 2; *// ListNotMem***short** StackError;  
  
**typedef** List Stack;  
  
**void** InitStack(Stack\* s);  
**void** PutStack(Stack\* s, BaseType e);  
**void** GetStack(Stack\* s, BaseType\* e);  
**short** EmptyStack(Stack\* s);  
**void** DoneStack(Stack\* s);  
  
  
#endif *//ASD\_6\_STACK\_H*

}

**Stack.c**

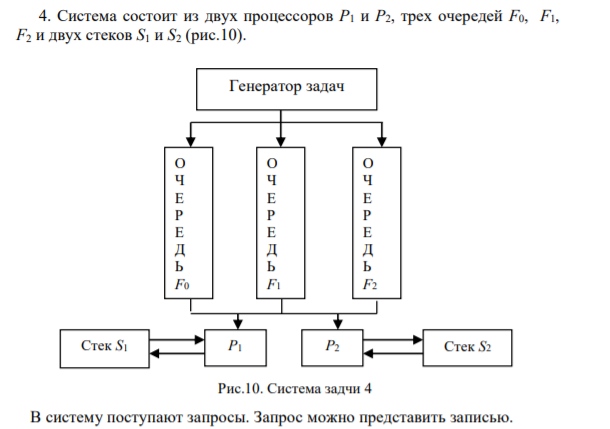
#include **"Stack.h"**#include **<stdio.h>**#include **<stdlib.h>**#include **"Stack.h"  
void** InitStack(Stack\* s)  
{  
 InitList(s);  
 StackError = ListError;  
}  
  
**void** PutStack(Stack\* s, BaseType e)  
{  
 PutList(s, e);  
 StackError = ListError;  
}  
  
**void** GetStack(Stack\* s, BaseType\* e)  
{  
 GetList(s, e);  
 StackError = ListError;  
}  
  
**short** EmptyStack(Stack\* s)  
{  
 **return** EmptyList(s);  
}  
  
**void** DoneStack(Stack\* s)  
{  
 DoneList(s);  
 StackError = ListError;  
}

**Fifo.h**

#ifndef **ASD\_6\_FIFO\_H**#define **ASD\_6\_FIFO\_H**#include **"List.h"  
  
static const short** FifoOk = 0; *// ListOk***static const short** FifoEmpty = 1; *// ListEmpty***static const short** FifoNotMem = 2; *// ListNotMem***short** FifoError;  
  
**typedef** List Fifo;  
  
**void** InitFifo(Fifo\* f);  
**void** PutFifo(Fifo\* f, BaseType e);  
**void** GetFifo(Fifo\* f, BaseType\* e);  
**short** EmptyFifo(Fifo\* f);  
**void** DoneFifo(Fifo\* f);  
  
#endif *//ASD\_6\_FIFO\_H*

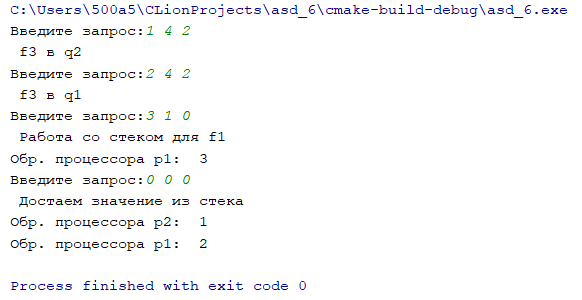
**Fifo.h**

#include **"Fifo.h"  
  
void** InitFifo(Fifo\* f)  
{  
 InitList(f);  
 FifoError = ListError;  
}  
  
**void** PutFifo(Fifo\* f, BaseType e)  
{  
 EndListPtr(f);  
 PutList(f, e);  
 FifoError = ListError;  
}  
  
**void** GetFifo(Fifo\* f, BaseType\* e)  
{  
 BeginListPtr(f);  
 GetList(f, e);  
 FifoError = ListError;  
}  
  
**short** EmptyFifo(Fifo\* f)  
{  
 **return** EmptyList(f);  
}  
  
**void** DoneFifo(Fifo\* f)  
{  
 DoneList(f);  
 FifoError = ListError;  
}



**Основная программа**

#include **<stdio.h>**#include **"Fifo.h"**#include **"Stack.h"  
  
  
  
int** main() {  
 Fifo\* f1 = (Fifo\*) malloc(**sizeof**(Fifo));  
 Fifo\* f2 = (Fifo\*) malloc(**sizeof**(Fifo));  
 Fifo\* f3 = (Fifo\*) malloc(**sizeof**(Fifo));  
  
 Stack\* s1 = (Stack\*) malloc(**sizeof**(Stack));  
 Stack\* s2 = (Stack\*) malloc(**sizeof**(Stack));  
  
 InitFifo(f1);  
 InitFifo(f2);  
 InitFifo(f3);  
 InitStack(s1);  
 InitStack(s2);  
 **short** p1 = 0, p2 = 0;  
  
 **int** k1 = 0, k2 = 0;  
  
 BaseType q, q1, q2;  
  
 **int** k = 0, n = 1;  
  
 **int** f = 0;  
  
 printf(**"Введите запрос: "**);  
 scanf(**"%d %d %d"**, &(q.data), &(q.time), &(q.p));  
  
 **while** (k < n)  
 {  
 **if**(!f)  
 {  
 **if**(q.p == 0)  
 {  
 PutFifo(f1, q);  
 }  
 **else if**(q.p == 1){  
 PutFifo(f2, q);  
 }  
 **else**{  
 PutFifo(f3, q);  
 }  
  
 }  
  
 **if** ((!EmptyFifo(f1) && !p1) || (!EmptyFifo(f1) && !p2) )  
 {  
 **if**(!p1)  
 {  
 printf(**"f1 в q1\n"**);  
 GetFifo(f1, &q1);  
 k1 = 0;  
 p1 = 1;  
 }  
 **else**{  
 printf(**"f1 в q2\n"**);  
 GetFifo(f1, &q2);  
 k2 = 0;  
 p2 = 1;  
 }  
 }  
  
 **if** (EmptyFifo(f1) && ((!EmptyFifo(f2) && !p1) || (!EmptyFifo(f2) && !p2)))  
 {  
 **if**(!p1)  
 {  
 printf(**"f2 в q1\n"**);  
 GetFifo(f2, &q1);  
 k1 = 0;  
 p1 = 1;  
 }  
 **else**{  
 printf(**"f2 в q2\n"**);  
 GetFifo(f2, &q2);  
 k2 = 0;  
 p2 = 1;  
 }  
 }  
  
 **if** (EmptyFifo(f1) && EmptyFifo(f2) && ((!EmptyFifo(f3) && !p1) || (!EmptyFifo(f3) && !p2)))  
 {  
 **if**(!p2)  
 {  
 printf(**"f3 в q2\n"**);  
 GetFifo(f3, &q2);  
 k2 = 0;  
 p2 = 1;  
 }  
 **else**{  
 printf(**"f3 в q1\n"**);  
 GetFifo(f3, &q1);  
 k1 = 0;  
 p1 = 1;  
 }  
 }  
  
 **if** (!EmptyFifo(f1) && p1 && p2 && (q1.p != 0 || q2.p != 0))  
 {  
 printf(**"Работа со стеком для f1\n"**);  
 **if**(q1.p != 0)  
 {  
 PutStack(s1, q1);  
 GetFifo(f1, &q1);  
 k2 = 0;  
 }  
 **else**{  
 PutStack(s1, q2);  
 GetFifo(f1, &q2);  
 k2 = 0;  
 }  
 }  
 **else if**(!EmptyFifo(f2) && p1 && p2 && (q1.p == 2 || q2.p == 2))  
 {  
 printf(**"Работа со стеком для f2\n"**);  
 **if**(q1.p == 2)  
 {  
 PutStack(s2, q1);  
 GetFifo(f2, &q1);  
 k2 = 0;  
 }  
 **else**{  
 PutStack(s2, q2);  
 GetFifo(f2, &q2);  
 k2 = 0;  
 }  
 }  
 **else if**(!(EmptyStack(s1)|| !EmptyStack(s2))&& (!p1 || !p2))  
 {  
 printf(**"Достаем значение из стека\n"**);  
 **if**(!p1)  
 {  
 GetStack(s1, &q1);  
 k1 = 0;  
 p1 = 1;  
 }  
 **else**{  
 GetStack(s2, &q2);  
 k2 = 0;  
 p2 = 1;  
 }  
  
 }  
  
 **if** (p1)  
 {  
 k1++;  
 **if** (k1 >= q1.time)  
 {  
 printf(**"Обр. процессора p1: %d \n"**, q1.data);  
 k1 = 0;  
 p1 = 0;  
 k++;  
 }  
 }  
  
 **if** (p2)  
 {  
 k2++;  
 **if** (k2 >= q2.time)  
 {  
 printf(**"Обр. процессора p2: %d \n"**, q2.data);  
 k2 = 0;  
 p2 = 0;  
 k++;  
 }  
 }  
  
 **if**(!f)  
 {  
 printf(**"Введите запрос: "**);  
 scanf(**"%d %d %d"**, &(q.data), &(q.time), &(q.p));  
 n++;  
  
 }  
 **if** (!f && q.data == 0)  
 {  
 f = 1;  
 n--;  
 }  
 }  
  
 **return** 0;  
}



**hash\_table.h**

#ifndef **ASD\_6\_HASH\_TABLE\_H**#define **ASD\_6\_HASH\_TABLE\_H  
static const short** TableOk = 0;  
**static const short** TableNotMem = 1;  
**static const short** TableNotKey = 2;  
**static const short** TableEmpty = 3;  
  
**short** TableError;  
  
**typedef int** KeyType;  
**typedef int** BaseType;  
  
**typedef int** (\*hash)(**int**);  
  
**typedef struct** eltable  
{  
 KeyType key;  
 BaseType e;  
 **struct** eltable\* next;  
};  
  
**typedef struct**{  
 hash h;  
 **struct** eltable\*\* buf;  
 **unsigned** n;  
 **unsigned** sizeBuf;  
} Table;  
  
**void** InitTable(Table\* t, **unsigned** sizeBuf, hash h);  
**int** PutTable(Table\* t, KeyType key, BaseType e);  
**int** GetTable(Table\* t, KeyType key, BaseType\* e);  
**int** EmptyTable(Table\* t);  
**void** DoneTable(Table\* t);  
  
#endif *//ASD\_6\_HASH\_TABLE\_H*

**hash\_table.с**

#include **<stdlib.h>**#include **"hash\_table.h"  
void** InitTable(Table\* t, **unsigned** sizeBuf, hash h)  
{  
 t->h = h;  
 t->sizeBuf = sizeBuf;  
 t->n = 0;  
 t->buf = (**struct** eltable\*\*) malloc(sizeBuf \* **sizeof**(**struct** eltable\*));  
  
 **if** (t->buf != **NULL**)  
 {  
 **for**(**int** i = 0; i < sizeBuf; i++)  
 t->buf[i] = **NULL**;  
 TableError = TableOk;  
 }  
 **else** {  
 TableError = TableNotMem;  
 }  
}  
  
**int** PutTable(Table\* t, KeyType key, BaseType e)  
{  
 **int** k = t->h(key);  
  
 **struct** eltable\* el = (**struct** eltable\*) malloc(**sizeof**(**struct** eltable));  
 **if** (el == **NULL**)  
 {  
 TableError = TableNotMem;  
 **return** 0;  
 }  
 el->key = key;  
 el->e = e;  
  
 el->next = t->buf[k];  
 t->buf[k] = el;  
 t->n++;  
  
 TableError = TableOk;  
 **return** 1;  
}  
  
**int** GetTable(Table\* t, KeyType key, BaseType\* e)  
{  
 **if** (EmptyTable(t))  
 {  
 TableError = TableEmpty;  
 **return** 0;  
 }  
  
 **int** k = t->h(key);  
  
 **struct** eltable\* el = t->buf[k];  
 **if** (el == **NULL**)  
 {  
 TableError = TableNotKey;  
 **return** 0;  
 }  
  
 **struct** eltable\* pr = **NULL**;  
 **while** (el->next != **NULL** && el->key != key)  
 {  
 pr = el;  
 el = el->next;  
 }  
  
 **if** (el->key == key)  
 {  
 \*e = el->e;  
 **if**(pr == **NULL**)  
 t->buf[k] = **NULL**;  
 **else** pr->next = el->next;  
 free(el);  
 t->n--;  
  
 TableError = TableOk;  
 **return** 1;  
 }  
 **else** {  
 TableError = TableNotKey;  
 **return** 0;  
 }  
}  
  
**int** EmptyTable(Table\* t)  
{  
 **return** t->n == 0;  
}  
  
**void** DoneTable(Table\* t)  
{  
 **if** (EmptyTable(t))  
 {  
 TableError = TableOk;  
 **return**;  
 }  
  
 **int** i = 0;  
 **while** (i < t->sizeBuf)  
 {  
 **struct** eltable\* el = t->buf[i];  
 **while** (el != **NULL**)  
 {  
 t->buf[i] = el->next;  
 free(el);  
 el = t->buf[i];  
 t->n--;  
 }  
 i++;  
 }  
  
 free(t->buf);  
  
 TableError = TableOk;  
}