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 автоматизированных систем

Дисциплина: Алгоритмы и структуры данных

Расчетно-графическое задание

|  |  |
| --- | --- |
|  | Выполнила: ст. группы ПВ-21  Зановская А.И.  Проверил: Синюк В.Г. |

Белгород

2017

**Стек как отображение на массив**

Заголовочный файл

#ifndef \_STACK\_ARRAY\_H\_INCLUDED

#define \_STACK\_ARRAY\_H\_INCLUDED

const short StackSize = 100;

const short stackOk = 0;

const short stackFull = 1;

const short stackEmpty = 2;

extern short StackError = 0;

typedef int BaseType;

typedef struct

{

BaseType buf[StackSize];

short uk;

} Stack;

void StackInit (Stack \*S);

void StackPut (Stack \*S, BaseType E);

void StackGet (Stack \*S, BaseType \*E);

int StackFull (Stack \*S);

int StackEmpty (Stack \*S);

#endif

Функции для работы со стеком

#include <stdio.h>

#include "Stack\_Array.h"

int StackError = 0;

//Инициализация стека

void StackInit(Stack \*S)

{

S->uk = 0;

StackError = stackOk;

}

//Включение в стек

void StackPut(Stack \*S, BaseType E)

{

if (StackFull(S))

{

StackError=stackFull;

return;

}

S->buf[S->uk] = E;

S->uk++;

StackError=stackOk;

}

//Исключение из стека

void StackGet(Stack \*S, BaseType \*E)

{

if (StackEmpty(S))

{

StackError=stackEmpty;

return;

}

\*E = S->buf[S->uk - 1];

S->uk--;

StackError=stackOk;

}

//Стек полон

int StackFull(Stack \*S)

{

return (S->uk == StackSize);

}

//Стек пуст

int StackEmpty(Stack \*S)

{

return (S->uk == 0);

}

**Очередь как отображение на массив**

Заголовочный файл

#ifndef \_FIFOARRAY\_H\_INCLUDED

#define \_FIFOARRAY\_H\_INCLUDED

const short FifoSize = 100;

const int fifoOk = 0;

const int fifoFull = 1;

const int fifoEmpty = 2;

extern short FifoError;

typedef int BaseType;

typedef struct

{

BaseType buf[FifoSize];

int uk1; // Указатель на хвост

int uk2; // Указатель на голову

int n; // Количество элементов в очереди

} Fifo;

void FifoInit (Fifo \*F);

void FifoPut (Fifo \*F, BaseType E);

void FifoGet (Fifo \*F, BaseType \*E);

int FifoEmpty (Fifo \*F);

int FifoFull (Fifo \*F);

#endif

Функции для работы с очередью

#include <stdio.h>

#include "FifoArray.h"

int FifoError;

//Инициализация очереди

void FifoInit (Fifo \*F)

{

F->uk1 = 0;

F->uk2 = 0;

F->n = 0;

FifoError = fifoOk;

}

//Включение в очередь

void FifoPut(Fifo \*F, BaseType E)

{

if (FifoFull(F))

{

FifoError=fifoFull;

return;

}

F->buf[F->uk2] = E;

F->uk2 = (F->uk2 + 1) % Fif0Size;

F->n++;

FifoError = fifoOk;

}

//Исключение из очереди

void FifoGet (Fifo \*F, BaseType \*E)

{

if (FifoEmpty(F))

{

FifoError=fifoEmpty;

return;

}

\*E = F->buf[F->uk1];

F->uk1 = (F->uk1 + 1) % FifoSize;

F->n--;

FifoError = fifoOk;

}

//Очередь полна

int FifoFull(Fifo \*F) {

return (F->n == FifoSize);

}

//Очередь пуста

int FifoEmpty(Fifo \*F){

return (F->len == 0);

}

**Односвязный линейный список**

Заголовочный файл

#ifndef \_LIST\_H\_INCLUDED

#define \_LIST\_H\_INCLUDED

const short ListOk=0;

const short ListNotMem=1;

const short ListEmpty=2;

static short ListError;

typedef int BaseType;

typedef struct element \*ptrel;

typedef struct element {

BaseType data;

ptrel next;

} element;

typedef struct{

ptrel Start;

ptrel ptr;

int N;

} List;

void InitList(List \*L)

void PutList(List \*L, BaseType E)

void GetList(List \*L, BaseType \*E)

void ReadList(List \*L, BaseType \*E)

int EmptyList(List \*L)

int EndList(List \*L)

unsigned int Count(List \*L)

void BeginPtr(List \*L)

void MovePtr(List \*L)

void MoveTo(List \*L, unsigned int n)

void DoneList(List \*L)

void CopyList(List \*L1, List \*L2)

#endif

Функции для работы с односвязным линейным списком

#include <stdio.h>

#include <stdlib.h>

#include "List.h"

//Инициализация списка

void InitList(List \*L)

{

L->Start = (ptrel)malloc(sizeof(element));

if (L->Start==NULL)

{

ListError=ListNotMem;

return;

}

L->ptr=L->Start;

L->ptr->next=NULL;

ListError=ListOk;

L->N=0;

}

//Вставка в список

void PutList(List \*L, BaseType E)

{

ptrel tmp = (ptrel)malloc(sizeof(element));

tmp->data=E;

tmp->next=L->ptr->next;

L->ptr->next=tmp;

L->ptr=L->ptr->next;

L->N+=1;

}

//Взятие из списка

void GetList(List \*L, BaseType \*E)

{

ptrel tmp;

if (EmptyList(L))

return;

if (EndList(L))

{

\*E=L->ptr->data;

free(L->ptr);

L->Start=NULL;

ListError=ListEmpty;

return;

}

tmp=L->ptr->next;

L->ptr->next=tmp->next;

\*E = tmp->data;

free(tmp);

}

//Чтение из списка

void ReadList(List \*L, BaseType \*E)

{

if (EmptyList(L))

{

ListError=ListEmpty;

return;

}

\*E=L->ptr->data;

ListError=ListOk;

}

//Предикат - список пуст

int EmptyList(List \*L)

{

return (L->Start->next == NULL);

}

//Предикат - конец списка

int EndList(List \*L)

{

return (L->ptr->next == NULL);

}

//Колличество элементов в списке

unsigned int Count(List \*L)

{

unsigned counter = 0;

ptrel tmp = L->Start->next;

while (tmp->next != NULL)

{

counter++;

tmp=tmp->next;

}

return counter;

}

//Перемещение текущего указателя в начало списка

void BeginPtr(List \*L)

{

L->ptr=L->Start;

}

//Перемещение текущего указателя на один эелемент вперёд

void MovePtr(List \*L)

{

L->ptr=L->ptr->next;

}

//Перемещение текущего указателя на позицию n

void MoveTo(List \*L, unsigned int n)

{

unsigned int counter=0;

ptrel tmp = L->Start;

while(counter++!=n)

tmp = tmp->next;

L->ptr=tmp;

}

//Очистка спика

void DoneList(List \*L)

{

BaseType e;

BeginPtr(L);

while (!EmptyList(L))

GetList(L,&e);

free(L->ptr);

ListError=ListEmpty;

L->N=0;

}

//Копирование первого списка во второй

void CopyList(List \*L1, List \*L2)

{

ptrel tmp = L1->Start;

while (tmp->next!=NULL)

{

PutList(L2, tmp->next->data);

tmp=tmp->next;

}

L2->N=L1->N;

}

**Стек как отображение на односвязный линейный список**

#ifndef \_STACK\_LIST\_H\_INCLUDED

#define \_STACK\_LIST\_H\_INCLUDED

#include "List.h"

const int StackOk = ListOk;

const int StackNotMem = ListNotMem;

const int StackEmpty = ListEmpty;

extern int StackError;

typedef List Stack;

void StackInit(Stack \*S);

void StackPut(Stack \*S, BaseType E);

void StackGet(Stack \*S, BaseType \*E);

void StackDone(Stack \*S);

int StackEmpty (StackList \*S);

#endif

Функции для работы со стеком

#include "Stack\_List.h"

#include "List.h"

//Инициализация стека

void StackInit(Stack \*S)

{

InitList(S);

StackError = ListError;

}

//Включение в стек

void StackPut(Stack \*S, BaseType E)

{

PutList(S, E);

StackError = ListError;

}

//Исключение из стека

void StackGet(Stack \*S, BaseType\*E)

{

GetList(S, E);

StackError = ListError;

}

//Удаление стека

void StackDone(Stack \*S)

{

DoneList(S);

StackError = ListError;

}

//Стек пуст

int StackEmpty(Stack \*S)

{

return EmptyList(S);

}

**Очередь как отображение на односвязный линейный список**

#ifndef \_FIFO\_LIST\_H\_INCLUDED

#define \_FIFO\_ LIST\_H\_INCLUDED

#include "List.h"

const int fifoOk = ListOk;

const int fifoEmpty = ListEmpty;

const int fifoNtMem = ListNotMem;

extern int ListError;

typedef int BaseType;

typedef List Fifo;

void FifoInit(Fifo \*S);

void FifoPut(Fifo \*S, BaseType E);

void FifoGet(Fifo \*S, BaseType \*E);

void FifoDone(Fifo \*L);

int FifoEmpty(Fifo \*S);

#endif

Функции для работы со стеком

#include "Fifo\_List.h"

//Инициализация очереди

void FifoInit(Fifo \*S)

{

InitList(S);

FifoError = ListError;

}

//Включение в очередь

void FifoPut (Fifo \*S, BaseType E)

{

MoveTo(S,S->N);

PutList(S, E);

FifoError = ListError;

}

//Исключение из очереди

void FifoGet (Fifo \*S, BaseType \*E)

{

BeginPtr(S);

GetList(S, E);

FifoError = ListError;

}

//Удаление очереди

void FifoDone (Fifo \*L)

{

DoneList(L);

}

//Очередь пуста

int FifoEmpty (Fifo \*S)

{

return EmptyList(S);

}

**Двусвязный линейный список**

#ifndef DLIST\_H

#define DLIST\_H

#include <stdio.h>

#include <stdlib.h>

const short DListOk = 0;

const short DListEmpty = 1;

const short DListNotMem = 2;

const short DListEnd = 3;

const short DListBegin = 4;

extern short DListError;

typedef int BaseType;

typedef struct element \* elptr;

typedef struct element{

BaseType Data;

elptr Llink;

elptr Rlink;

} element ; //элемент списка

typedef struct {

elptr L;

elptr R;

elptr ptr;

} DList;//дескриптор

void DListInit(DList\* DL); //инициализация списка

void DListDone(DList\* DL); //удаление списка

int isDListEmpty(DList\* DL); //список пуст

void DListPutPred(DList\* DL, BaseType E); //включение элемента в список перед рабочим указателем

void DListPutPost(DList\* DL, BaseType E); //включение элемента в список после рабочего указателя

void DListGetPred(DList\*DL, BaseType\* E); //исключение элемента из списка перед рабочим указателем

void DListGetPost(DList\*DL, BaseType\* E); //исключение элемента из списка после рабочего указателя

void DListL(DList\* DL); //передвинуть рабочий указатель влево

void DListR(DList\* DL); //передвинуть рабочий указатель вправо

void MoveDListBegin(DList\* DL); //установить указатель в начало списка

void MoveDListEnd(DList\* DL);// установить указатель в конец списка

int isDListEnd(DList\* DL); //конец списка

int isDListBegin(DList\* DL); //начало списка

#endif // DLIST\_H

**Функции для работы с двусвязным линейным списком**

#include "DList.h"

short DListError = 0;

void DListInit(DList\* DL) //инициализация списка

{

DL->L = NULL;

DL->R = NULL;

DL->ptr = NULL;

DListError = DListOk;

}

int isDListEmpty(DList\* DL) //список пуст

{

return DL->L==NULL;

}

void DListPutPred(DList\* DL, BaseType E) //включение элемента в список перед рабочим указателем

{

elptr pntr = (elptr) malloc(sizeof(element));

if (pntr == NULL)

{

DListError = DListNotMem;

return;

}

pntr -> Data = E;

if (isDListEmpty(DL)) //список пуст

{

pntr ->Llink = pntr -> Rlink = NULL;

DL->L=DL->R=DL->ptr = pntr;

}

else if (isDListBegin(DL)) //включение между дескриптором и рабочим указателем

{

pntr->Llink = DL->L;

pntr->Rlink = NULL;

DL->L->Rlink = pntr;

DL->L = pntr;

}

else //общий случай

{

pntr->Llink = DL->ptr;

pntr->Rlink = DL->ptr->Rlink;

DL->ptr->Rlink->Llink = pntr;

DL->ptr->Rlink = pntr;

}

DListError=DlistOk;

}

void DListPutPost(DList\* DL, BaseType E) //включение элемента в список после рабочего указателя

{

elptr pntr = (elptr) malloc(sizeof(element));

if (pntr == NULL)

{

DListError = DListNotMem;

return;

}

pntr -> Data = E;

if (isDListEmpty(DL)) //список пуст

{

pntr ->Llink = pntr -> Rlink = NULL;

DL->L=DL->R=DL->ptr = pntr;

}

else if (isDListEnd(DL)) //указатель в конце списка

{

pntr->Llink = NULL;

pntr->Rlink = DL->ptr;

DL->ptr->Llink = pntr;

DL->R = pntr;

}

else //общий случай

{

pntr->Llink = DL->ptr->Llink;

pntr->Rlink = DL->ptr;

DL->ptr->Llink->Rlink = pntr;

DL->ptr->Llink = pntr;

}

DListError=DlistOk;

}

void DListGetPred(DList\*DL, BaseType\* E) //исключение элемента из списка перед рабочим указателем

{

elptr pntr;

if (isDListEmpty(DL)) //список пуст

{

DListError = DListEmpty;

return;

}

if (isDListBegin(DL)) //указатель в начале списка

{

DListError = DListBegin;

return;

}

pntr = DL->ptr->Rlink;

(\*E) = pntr->Data;

if (pntr->Rlink == NULL) //удаляется первый элемент списка

DL->L = DL->ptr;

else

pntr->Rlink = DL->ptr;

DL->ptr->Rlink = pntr->Rlink;

free((void\*)pntr);

}

void DListGetPost(DList\*DL, BaseType\* E) //исключение элемента из списка после рабочего указателя

{

elptr pntr;

if (isDListEmpty(DL)) //список пуст

{

DListError = DListEmpty;

return;

}

if (isDListEnd(DL)) //указатель в конце списка

{

DListError = DListEnd;

return;

}

pntr = DL->ptr->Llink; //удаляемый элемент

(\*E) = pntr->Data;

if (pntr->Llink==NULL) //если удаляемый элемент последний

DL->R = DL->ptr;

else

pntr->Llink->Rlink = DL->ptr;

DL->ptr->Llink = pntr->Llink;

free((void\*)pntr);

}

void DListL(DList\* DL) //передвинуть рабочий указатель влево

{

if (isDListEmpty(DL))

{

DListError = DListEmpty;

return;

}

if (!isDListBegin(DL))

DL->ptr = DL->ptr->Rlink;

else

{

DListError = DListBegin;

return;

}

DListError = DListOk;

}

void DListR(DList\* DL) //передвинуть рабочий указатель вправо

{

if (isDListEmpty(DL))

{

DListError = DListEmpty;

return;

}

if (!isDListEnd(DL))

DL->ptr = DL->ptr->Llink;

else

{

DListError = DListEnd;

return;

}

DListError = DListOk;

}

int isDListEnd(DList\* DL) //конец списка

{

return DL->ptr == DL->R;

}

int isDListBegin(DList\* DL) //начало списка

{

return DL->ptr == DL->L;

}

void MoveDListEnd(DList\* DL) // установить указатель в конец списка

{

if(isDListEmpty(DL))

{

DListError = DListEmpty;

return;

}

DL->ptr = DL->R;

DListError=DlistOk;

}

void MoveDListBegin(DList\* DL) // установить указатель в начало списка

{

if(isDListEmpty(DL))

{

DListError = DListEmpty;

return;

}

DL->ptr = DL->L;

DListError=DlistOk;

}

void DListDone(DList\* DL) //сделать список пустым

{

BaseType E;

if (!isDListEmpty(DL))

{

MoveDListBegin(DL);

while (!isDListEnd(DL))

DListGetPost(DL, &E);

free((void\*)DL->ptr);

DL->L = NULL;

DL->R = NULL;

DL->ptr = NULL;

}

DListError = DListEmpty;

}

**Дек как отображение на двусвязный линейный список**

#ifndef \_DECK\_H\_

#define \_DECK\_H\_

#include "DList.h"

const int DeckOk = DListOk;

const int DeckNotMem = DListNotMem;

const int DeckEmpty = DListEmpty;

extern int DeckError;

typedef int BaseType;

typedef DList Deck;

void DeckInit(Deck \*D);

void DeckDone(Deck \*D);

int DeckEmpty(Deck \*D);

void DeckPutE(Deck \*D, BaseType E);

void DeckPutB(Deck \*D, BaseType E);

void DeckGetE(Deck \*D, BaseType \*E);

void DeckGetB(Deck \*D, BaseType \*E);

#endif

**Функции для работы с деком**

#include "Deck.h"

//Инициализация дека

void DeckInit(Deck \*D)

{

DListInit(D);

DeckError = DListError;

}

//Удаление дека

void DeckDone(Deck \*D)

{

DListDone(D);

DeckError = DListError;

}

//дек пуст

int DeckEmpty(Deck \*D)

{

return DListEmpty(D);

}

//Включение в начало дека

void DeckPutB(Deck \*D, BaseType E)

{

DListBegin(D);

DListPutPost(D, E);

DeckError = DListError;

}

//Включение в конец дека

void DeckPutE(Deck \*D, BaseType E)

{

DLsistEnd(D);

DListPutPred(D, E);

DeckError = DListError;

}

//Исключение из начала дека

void DeckGetB(Deck \*D, BaseType \*E){

DListBegin(D);

DListGetPost(D, E);

DeckError = DListError;

}

//Исключение из конца дека

void DeckGetE(Deck \*D, BaseType \*E){

DListE(D);

DListGetPred(D, E);

DeckError = DListError;

}

**Очередь с приоритетом**

#ifndef \_FIFOPR\_H\_

#define \_FIFOPR\_H\_

#define FIFOSIZE 20

const int FifoPrOk = 0;

const int FifoPrFull = 1;

const int FifoPrEmpty = 2;

extern int FifoPrError;

typedef struct {

int data;

int pr;

} BaseType;

typedef struct {

BaseType buf[FIFOSIZE];

unsigned uk;

} FifoPr;

void FifoPrInit(FifoPr \*F);

void FifoPrPut(FifoPr \*F, BaseType E);

void FifoPrGet(FifoPr \*F, BaseType \*E);

int FifoPrFull(FifoPr \*F);

int FifoPrEmpty(FifoPr \*F);

#endif

**Функции для работы с очередью с приоритетом**

#include <stdio.h>

#include "FifoPr.h"

int FifoPrError;

//Инициализация

void FifoPrInit(FifoPr \*F)

{

F->uk = 0;

FifoPrError = FifoPrOk;

}

//Включение в очередь

void FifoPrPut(FifoPr \*F, BaseType E)

{

if (FifoPrFull(F))

{

FifoPrError = FifoPrFull;

return;

}

F->buf[F->uk] = E;

F->uk++;

FifoPrError = FifoPrOk;

}

//Исключение

void FifoPrGet(FifoPr \*F, BaseType \*E)

{

if (FifoPrEmpty(F))

{

FifoPrError = FifoPrEmpty;

return;

}

BaseType max = F->buf[0];

int maxPos = 0,i;

for (i=1;i<F->uk;i++)

{

if (F->buf[i].priority > max.priority) {

max = F->buf[i];

maxPos = i;

}

}

\*E = max;

F->buf[maxPos] = F->buf[--F->uk];

FifoPrError = FifoPrOk;

}

//Очередь полна

int FifoPrFull(FifoPr \*F) {

return (F->uk == FIFOSIZE);

}

//Очередь пуста

int FifoPrEmpty(FifoPr \*F) {

return (F->uk == 0);

}

**Односвязный циклический линейный список**

Заголовочный файл

#ifndef \_СLIST\_H\_

#define \_СLIST\_H\_

#include "List.h"

const int CListOk = ListOk;

const int CListNotMem = ListNotMem;

const int CListEmpty = ListEmpty;

extern int CListError;

typedef List CList;

void CListInit(Stack \*S);

void CListGet(Stack \*S, BaseType \*E);

void CopyCList(Stack \*S);

void PutCList(List \*L, BaseType E);

void ReadCList(List \*L, BaseType \*E);

int EmptyCList(List \*L);

void CListBeginPtr(List \*L);

void CListMovePtr(List \*L);

void CListMoveTo(List \*L, unsigned int n);

void DoneCList(List \*L);

int StackEmpty (StackList \*S);

#endif

**Функции для работы с односвязным циклическим линейным списком**

#include <stdio.h>

#include <stdlib.h>

#include "CList.h"

//Инициализация списка

void InitListC (CList \*L, BaseType E)

{

L->ptr = NULL;

L->start = NULL;

L->N = 0;

}

//Список пуст

int IsEmptyListC(CList \*L)

{

return (L->Start!=NULL);

}

//Конец списка

int IsEndListC(CList \*L)

{

return (L->ptr->next==L->Start);

}

//Перенос указателя в начало списка

void BeginPtrC(CList \*L)

{

L->ptr=L->start;

}

//Включение в список

void PutListC (CList \*L, BaseType E)

{

ptrel tmp = (ptrel) malloc(sizeof(element));

if (tmp == NULL)

{

CListError = CListNotMem;

return;

}

tmp->data=E;

if(IsEmptyListC(L))

{

L->ptr->next=L->ptr=L->start=tmp;

CListError = CListNotMem;

}

else

{

tmp -> next = L -> ptr -> next;

L -> ptr -> next = tmp;

}

L->N++;

}

void GetListC (CList \*L, BaseType \*E)

{

if (IsEmptyListC(L)) return;

if(L->start->next==L->start)

{

\*E = L -> start -> data;

free( L -> start );

L -> start = L -> ptr = NULL;

CListError = CListEmpty;

}

else

{

if(IsEndListC(L)) L->start=L->ptr;

ptrel tmp = L -> ptr -> next;

\*E = tmp -> data;

L -> ptr -> next = tmp-> next;

free(tmp);

}

L->N--;

CListError = CListOk;

}

void DoneListС(СList \*L)

{

BaseType Del;

while (L->N) GetListC(L, &Del);

CListError = CListEmpty;

}

void MovePtrListC(CList \*L)

{

L->ptr=L->ptr->next;

}

**Таблица как отображение на массив**

Заголовочный файл

#ifndef \_TABLE\_ARRAY\_H\_

#define \_TABLE\_ARRAY\_H\_

#define TABLESIZE 30

const int tableOk = 0;

const int tableEmpty = 1;

const int tableFull = 2;

extern int TableError;

typedef int TableBaseType;

typedef int KeyBaseType;

typedef struct

{

TableBaseType data;

KeyArrayBaseType key;

} element;

typedef struct {

element buf[TABLESIZE];

unsigned uk;

} Table;

void TableInit(Table \*T);

void TablePut(Table \*T, element E);

void TableGet(Table \*T, element \*E, KeyBaseType key);

int searchKey(Table \*T, KeyBaseType key);

int TableEmpty(Table \*T);

int TableFull(Table \*T);

#endif

**Функции для работы с таблицей**

#include <stdio.h>

#include "Table\_Array.h"

int TableError;

//Инициализация таблицы

void TableInit(Table \*T)

{

T->uk = 0;

TableError = tableOk;

}

//Включение элемента в таблицу

void TablePut(Table \*T, element E)

{

if (TableFull(T))

{

TableError=tableFull;

return;

}

int pos;

if ((pos = searchKey(T, E.key)) != -1)

T->buf[pos].data = E.data;

else

{

T->buf[T->uk] = E;

T->uk++;

}

TableError=tableOk;

}

//Исключение элемента из таблицы

void TableGet(Table \*T, element \*E, KeyBaseType key)

{

if (TableEmpty(T))

{

TableError=tableEmpty;

return;

}

int pos;

if ((pos = searchKey(T, key)) != -1)

{

E->data = T->buf[pos].data;

E->key = T->buf[pos].key;

if (pos != T->uk - 1)

T->buf[pos] = T->buf[T->uk - 1];

T->uk--;

}

TableError=tableOk;

}

//Поиск элемента с данным ключом

int searchKey(Table \*T, KeyBaseType key)

{

unsigned i;

for (i=0; i<T->uk&&T->buf[i].key!=key;i++);

if (i == T->uk)

{

TableError = tableNoKey;

return -1;

}

return i;

}

//Таблица пуста

int TableEmpty (Table \*T)

{

return (T->uk == 0);

}

//Таблица переполнена

int TableFull(Table \*T)

{

return (T->uk == TABLESIZE);

}

**Таблица как отображение на ОЛС**

Заголовочный файл

#ifndef \_TABLE\_LIST\_H\_

#define \_TABLE\_LIST\_H\_

#include "List.h"

const int tableOk = ListOk;

const int tableEmpty = ListEmpty;

extern int TableListError;

typedef int TableBaseType;

typedef int KeyBaseType;

typedef struct element \*ptrel;

typedef struct

{

TableBaseType data;

KeyArrayBaseType key;

ptrel next;

} element;

typedef struct

{

ptrel Start;

ptrel ptr;

} Table;

void TableListInit(Table \*T);

void TableListPut(Table \*T, element E);

void TableListGet(Table \*T, element \*E, KeyBaseType key);

int TableListEmpty(Table \*T);

#endif

**Функции для работы с таблицей**

#include <stdio.h>

#include "Table\_List.h"

int TableListError;

//Инициализация таблицы

void TableInit(Table \*T)

{

ListInit(T);

TableListError=ListError;

}

//Включение элемента в таблицу

void TablePut(Table \*T, element \*E)

{

if (TableFull(T))

{

TableListError=TableListFull;

return;

}

ptrel tmp = (ptrel)malloc(sizeof(element));

tmp->data=E->data;

tmp->key=E->key;

tmp->next=L->ptr->next;

L->ptr->next=tmp;

L->ptr=L->ptr->next;

TableListError=TableListOk;

}

//Исключение элемента из таблицы

void TableGet(Table \*T, element \*E, KeyBaseType key)

{

if (TableEmpty(T))

{

TableListError=TableListEmpty;

return;

}

ptrel tmp;

tmp=L->ptr->next;

L->ptr->next=tmp->next;

E->data = tmp->data;

E->key = tmp->key;

free(tmp);

}

//Таблица пуста

int TableEmpty (Table \*T)

{

return ListEmpty(L);

}