**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

Лабораторная работа №5

дисциплина: Алгоритмы и Структуры данных

тема: «Структуры данных «линейные списки» (Pascal/С)»

Выполнил: ст. группы ПВ-21

Донцов Александр Алексеевич

Проверил: Синюк Василий Григорьевич

Белгород 2018

**Цель работы:** изучить СД типа «линейный список», научиться их программно реализовывать и использовать.

**Задание**

1. Для СД типа «линейный список» определить:

1.1. Абстрактный уровень представления СД:

1.1.1. Характер организованности и изменчивости.

1.1.2. Набор допустимых операций.

1.2. Физический уровень представления СД:

1.2.1. Схему хранения.

1.2.2. Объем памяти, занимаемый экземпляром СД.

1.2.3. Формат внутреннего представления СД и способ его интерпретации.

1.2.4. Характеристику допустимых значений.

1.2.5. Тип доступа к элементам.

1.3. Логический уровень представления СД.

Способ описания СД и экземпляра СД на языке программирования.

2. Реализовать СД типа «линейный список» в соответствии с вариантом индивидуального задания (см. табл.14) в виде модуля.

3. Разработать программу для решения задачи в соответствии с вариантом индивидуального задания (см. табл.14) с использованием модуля, полученного в результате выполнения пункта 2 задания.



**Отчёт**

1. Для СД типа «линейный список» определить:

1.1. Абстрактный уровень представления СД:

1.1.1. Характер организованности и изменчивости.

Последовательность

1.1.2. Набор допустимых операций.

Операции доступа

Возврат по индексу

Запись по индексу

Выделение памяти по индексу

Очищение памяти по индексу

1.2. Физический уровень представления СД:

1.2.1. Схему хранения.

связанная

1.2.2. Объем памяти, занимаемый экземпляром СД.

1.2.3. Формат внутреннего представления СД и способ его интерпретации.

1.2.4. Характеристику допустимых значений.

1.2.5. Тип доступа к элементам.

По указателю

1.3. Логический уровень представления СД.

Способ описания СД и экземпляра СД на языке программирования.

2. Реализовать СД типа «линейный список» в соответствии с вариантом индивидуального задания (см. табл.14) в виде модуля.

3. Разработать программу для решения задачи в соответствии с вариантом индивидуального задания (см. табл.14) с использованием модуля, полученного в результате выполнения пункта 2 задания.

**Код OLS.cpp**

#include "OLS.h"

void InitList(List \*\*L) {//инициализация списка+

\*L = (List\*)malloc(sizeof(List));

(\*L)->N = 0;

(\*L)->Start = NULL;

(\*L)->ptr = NULL;

}

void PutList(List \*L, BaseType E) {//включение элемента в список+

element\* pntr = (element\*)malloc(sizeof(el));

pntr->data = E;

if (L->Start == NULL) {//Установка первого элемента в списке

L->Start = pntr;

L->ptr = pntr;

pntr->next = NULL;

}

else {

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

L->ptr->next = pntr;

L->ptr = pntr;

}

L->N += 1;

printf("%lf\n", L->Start->data);

}

void GetList(List \*L, BaseType \*E) {//исключение элемента из списка

element\* pntr;

if (L->N == 0)

return;

else

if (L->N == 1) {

free(L->Start);

L->Start = NULL;

L->ptr = NULL;

}

else {

pntr = L->ptr;

BeginPtr(L);

while (L->ptr->next != pntr)

MovePtr;

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

free(pntr);

}

L->N -= 1;

}

void ReadList(List \*L, BaseType \*E) {//чтение элемента списка+

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

}

int EmptyList(List \*L) {//проверка: свободен ли список+

if (L->Start == NULL)

return 1;

else

return 0;

}

int EndList(List \*L) {//проверка: является ли элемент последним+

if (L->ptr->next == NULL)

return 1;

else

return 0;

}

unsigned int Count(List \*L) {//возвращает количество элементов в списке+

return L->N;

}

void BeginPtr(List \*L) {//устанановка в начало списка+

L->ptr = L->Start;

}

void EndPtr(List \*L) {//устанановка в конец списка+

while (L->ptr->next != NULL)

MovePtr(L);

}

void MovePtr(List \*L) {//переход к следующему элементу+

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

}

void MoveTo(List \*L, unsigned int n) {//переход к n-му элементу.

L->ptr = L->Start;

for (unsigned int i = 0; i < n; i++)

MovePtr(L);

}

void DoneList(List \*L) {//удаление списка.

element\* pntr;

BeginPtr(L);

while (L->ptr->next != NULL) {

pntr = L->ptr->next;

free(L->ptr);

L->ptr = pntr;

}

free(L->ptr);

free(L);

}

void CopyList(List \*L1, List \*L2) {//копирование списка L1 в список L2.

BeginPtr(L1);

BeginPtr(L2);

BaseType E;

while (EndList(L1)) {

ReadList(L1, &E);

PutList(L2, E);

MovePtr(L1);

}

}

Код OLS.h

#pragma once

#include <stdio.h>

#include <stdlib.h>

#ifndef \_\_LIST1\_H

#define \_\_LIST1\_H

const int ListOk = 0;

const int ListNotMem = 1;

const int ListUnder = 2;

const int ListEnd = 3;

typedef double BaseType;

struct el {

BaseType data;

struct el\* next;

};

typedef struct el element;

typedef struct {

element\* Start;

element\* ptr;

unsigned int N;

} List;

extern short ListError;

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 EndPtr(List \*L);

void MovePtr(List \*L);

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

void DoneList(List \*L);

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

#endif

Код Task.cpp

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include "OLS.h"

#include <math.h>

#include <locale.h>

#define EPS 0.0001

short ListError;

void inputPolynom(List \*L, int n);

double sumPolynom(List \*L, int x);

int equal(List \*p, List \*q, int x);

int main()

{

setlocale(LC\_ALL, "Rus");

int n = 3;

List\* L;

InitList(&L);

inputPolynom(L, n);

printf("\n\n---------%lf", sumPolynom(L, 10));

DoneList(L);

return 0;

}

void inputPolynom(List \*L ,int n) {

double a;

for (int i = 0; i < n; i++) {

printf("Введите %d коэффициент: ", i+1);

scanf("%lf", &a);

if (a != 0)

PutList(L, a);

}

return;

}

double sumPolynom(List \*L, int x) {

double sum = 0;

double \*path;

double data = x;

BeginPtr(L);

for (unsigned int i = 1; i <= L->N; i++) {

ReadList(L, &data);

MovePtr(L);

sum = x\*sum + data;

}

return sum;

}

int equal(List \*p, List \*q, int x) {

double sum1, sum2;

sum1 = sumPolynom(p, x);

sum2 = sumPolynom(q, x);

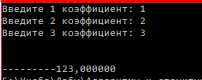
if ((sum1 - sum2) < EPS)

return 1;

else

return 0;

}



Двусвязный линейный список

#pragma once

#ifndef LIST\_H\_

#define LIST\_H\_

// константы ошибок

const short List\_OK = 0;

const short List\_Empty = 1;

const short List\_NotMem = 2;

const short List\_End = 2;

//переменная ошибок

extern short ListError;

//базовый тип

typedef int BaseType;

// элемент списка

typedef struct {

BaseType Data; // данные

element\* Next; // указатель на следующий элемент

element\* Back; // указатель на предыдущий элемент

} element;

// двухсвязный линейный список

typedef struct {

//указатель на первый элемент списка

element\* first;

//указатель на текущий элемент

element\* current;

} List;

//конструктор

void ListInit(List \*L);

//деструктор

void ListDone(List \*L);

//добавление элемента после рабочего указателя

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

//взятие элемента по рабочему указателю

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

// перенос рабочего указателя на следующий элемент

void ListMoveNext(List \*L);

// перенос рабочего указателя на предыдущий элемент

void ListMoveBack(List \*L);

//являеться ли список пустым

int ListEmpty(List \*L);

//находиться ли рабочий указатель в конце списка

int ListEnd(List \*L);

//перенос указателя в конец списка

void ListPtrEnd(List \*L);

//перенос указателя в начало списка

void ListPtrBegin(List \*L);

#endif

#include "DLS.h"

#include "stdlib.h"

short ListError = List\_OK;

//конструктор

void ListInit(List \*L) {

L->first = NULL;

L->current = NULL;

}

//деструктор

void ListDone(List \*L) {

element \* del;// удаляемый элемент

element \* next; // следующий элемент

//если список не пустой

if (L->first != NULL) {

next = L->first;

do {

del = next;

freeElem(del);

next = next->Next;

} while (next != NULL);

}

L->current = NULL;

L->first = NULL;

}

//создание и инициализация элемента списка

element \*creatElem(BaseType E) {

element \* el = (element \*)malloc(sizeof(element));

//если не удалось выделить память указателю

//возвращаем нулевой указатель

if (el == NULL)

return NULL;

el->Data = E;

el->Next = NULL;

el->Back = NULL;

return el;

}

//добавление элемента после рабочего указателя

void ListPut(List \*L, BaseType E) {

//создаем элемент списка

element \*el = creatElem(E);

//если неудалось выделить память

if (el == NULL){

//инициализируем переменную ошибки

ListError = List\_NotMem;

return;

}

//если список пуст, то вставляем в начало

if (ListEmpty(L)) {

//указатель СПИСКА на первый элемент указывает на созданный элемент

L->first = el;

//указатель СПИСКА на текущий элемент указывает на созданный элемент

L->current = el;

} else {//если список НЕ пуст, то вставляем после рабочего указателя

el->Next = L->current->Next;

L->current->Back = el;

L->current->Next = el;

}

ListError = List\_OK;

}

//очищение памяти элемента

void freeElem(element \*el) {

free(el);

}

//взятие(удаление) элемента по рабочему указателю

void ListGet(List \*L, BaseType \*E) {

//если список пустой, то инициализировать переменную ошибки

if (ListEmpty(L)) {

ListError = List\_Empty;

return;

}

//запоминаем данные текущего элемента

\*E = L->current->Data;

//УДАЛЯЕМ ТЕКУЩИЙ ЭЛЕМЕНТ

//1 ситуация - текущий элемент первый в списке

if (L->current->Back == NULL) {

L->first = L->current->Next;

L->first->Back = NULL;

freeElem(L->current);

L->current = L->first;

} else if (ListEnd(L)) {//2 ситуация - текущий элемент в конце

element \*tmpEl = L->current->Back;

tmpEl->Next = NULL;

freeElem(L->current);

L->current = tmpEl;

} else {//3 ситуация - текущий элемент где-то в центре

L->current->Next->Back = L->current->Back;

L->current->Back->Next = L->current->Next;

element \*tmpEl = L->current->Next;

freeElem(L->current);

L->current = tmpEl;

}

ListError = List\_OK;

}

// перенос рабочего указателя на следующий элемент

void ListMoveNext(List \*L) {

//если рабочий указатель стоит НЕ на последнем элементе, то передвигаем

if (L->current->Next != NULL)

L->current = L->current->Next;

}

// перенос рабочего указателя на предыдущий элемент

void ListMoveBack(List \*L) {

//если рабочий указатель НЕ на первом элементе, то передвигаем

if (L->current->Back != NULL)

L->current = L->current->Back;

}

//являеться ли список пустым

int ListEmpty(List \*L) {

return (L->first == NULL) ? 1 : 0;

}

//находиться ли рабочий указатель в конце списка

int ListEnd(List \*L) {

return (L->current->Next == NULL) ? 1 : 0;

}

//перенос указателя в конец списка

void ListPtrEnd(List \*L) {

//если список пустой, то инициализировать переменную ошибки

if (ListEmpty(L)) {

ListError = List\_Empty;

return;

}

while (L->current->Next != NULL) {

L->current = L->current->Next;

}

}

//перенос указателя в начала списка

void ListPtrBegin(List \*L) {

//если список пустой, то инициализировать переменную ошибки

if (ListEmpty(L)) {

ListError = List\_Empty;

return;

}

L->current = L->first;

}