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 №5

по дисциплине: «Алгоритмы и структуры данных»

по теме: **«Структуры данных «линейные списки» (C)»**

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

Белгород, 2024

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

Задания:

1. Для СД типа «линейный список» 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 «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.

Содержание

Задание 1: **3**
Абстрактный уровень: . . . 3
Задание 1.1.1: . . . 3
Задание 1.1.2: . . . 3
Физический уровень: . . . 4
Задание 1.2.1: . . . 4
Задание 1.2.2: . . . 4
Задание 1.2.3: . . . 4
Задание 1.2.4: . . . 5
Задание 1.2.5: . . . 5
Логический уровень: . . . 5

Задание 2: **8**

Задание 3: **24**

Вывод: **26**

Задание 1:

Опишем СД типа «линейный список».

Абстрактный уровень:

Задание 1.1.1:

СД типа «линейный список» является линейной СД (что, собственно, и отражено в её названии). Она, как и массив, отображает некую последовательность элементов: эти элементы находятся в линейном порядке, и для каждого из них, кроме первого и последнего, существует предыдущий и следующий. СД типа «линейный список» может быть реализована для любого типа элементов. Можно сказать, что на абстрактном уровне СД типа «линейный список» и СД типа массив схожи, различия появляются на физическом и логическом уровнях; обе СД хранят последовательности элементов, но выбраны разные способы их представления и сохранения упорядоченности с памяти компьютера.

Линейный список является динамической СД: в нем могут меняться как сами элементы, так и их количество. Причем, в отличие от массива, это количество не обязательно ограничено количеством элементов, под которые память была выделена при инициализации (это зависит от того, какая схема хранения была выбрана для реализации структуры; этот аспект подробнее рассмотрим в задании 1.2). К каждому элементу списка можно обратиться, зная его место в списке (индекс) и перезаписать его значение, однако процесс поиска элемента с данным индексом будет отличаться от такового в массиве (что, опять-таки, будет рассмотрено далее).

Задание 1.1.2:

В отличие от СД, рассматриваемых в прошлых лабораторных работах, СД типа «линейный список» является не встроенной, а производной, то есть не реализована в языках программирования по умолчанию. Тем не менее, для линейного списка определен набор обязательных операций:

1. Инициализация.
2. Включение элемента.
3. Исключение элемента.
4. Чтение текущего элемента.
5. Переход в начало списка.
6. Переход в конец списка.
7. Переход к следующему элементу.
8. Переход к i-му элементу.
9. Определение длины списка.
10. Уничтожение списка.

Физический уровень:

Задание 1.2.1:

Здесь мы подходим к основному отличию СД типа массив и СД типа «линейный список». В отличие от массива, который всегда имеет прямоугольную схему хранения, линейный список может быть реализован как при прямоугольной, так и при связной схеме хранения. С прямоугольной схемой мы уже знакомы: при ее использовании все элементы СД хранятся в памяти компьютера последовательно: последовательности битов, представляющие нулевой, первый, второй и так далее элемент, идут друг за другом по возрастанию адреса без разделителей. Линейный список, использующий эту схему, называется последовательным линейным списком (ПЛС). При связной же схеме хранения каждый элемент не просто хранит некоторое значение — часть последовательности, а является элементом типа «запись» (такие элементы также называют узлами), куда кроме непосредственно значения записаны еще и адреса других элементов списка. Список, реализованный с помощью этой схемы, называется связным линейным списком (СЛС); связные линейные списки делятся на группы и далее, в зависимости от количества указателей, которые содержит тип «запись». Если указатель один и для каждого элемента указывает на следующий/предыдущий элемент, то линейный список называется односвязным (ОЛС). Если указателей два, и один указывает на предыдущий, а второй — на следующий, то линейный список называется двусвязным (ДЛС). Если же каждый элемент содержит множество указателей на элементы, являющиеся частью последовательности, объединенной каким-либо признаком, то линейный список называется многосвязным. Связные линейные списки, тем не менее, также могут быть реализованы с использованием статической памяти и массивов.

Задание 1.2.2:

Количество памяти, занимаемой списком, зависит от того, какое количество памяти занимает его базовый тип в частности и тип «запись» (для СЛС) в целом и сколько элементов содержит список. Если базовый тип занимает T байт, указатель (типа *unsigned long*) — 8 байт, а список содержит K элементов, то для ОЛС количество памяти, занимаемой экземпляром, будет равно $(T + 8) * K$ байт, для ДЛС $(T + 16) * K$ байт, для не использующих указатели ПЛС $T * K$ байт.

Задание 1.2.3:

Если линейный список с длиной K является последовательным, то значение этой структуры хранится как K идущих подряд последовательностей битов, кодирующих значения элементов списка в соответствии с правилами для данного базового типа. Для связных же списков каждый узел хранится независимо друг от друга. Сами по себе узлы являются структурами, где одно поле отведено для хранения значения элемента (способ перевода в двоичный код зависит от базового типа), а одно, два или несколько полей — указателями на следующий узел (указатели кодируются как положительные целые числа — переводятся в двоичную систему счисления и дополняются нулями слева до размера в 8 байт). Двоичные коды значений отдельных полей структур также могут храниться в памяти компьютера не подряд (либо в порядке, отличном от заданного). Также связный линейный список может обладать фиктивными узлами, то есть узлами, где поле данных пусто, в его концах — одним узлом для ОЛС и двумя узлами для ДЛС.

Задание 1.2.4:

Диапазон допустимых значений линейного списка также связан с диапазоном допустимых значений его базового элемента. Однако всегда можно точно сказать, что количество возможных значений для линейного списка определенной длины K равно возведенному в степень K количеству возможных значений для его базового элемента. Соответственно, для СД типа «линейный список» количество возможных значений равно сумме количеств возможных значений списков с длинами от 0 до max, где max — максимальная длина списка (однозначно определенная в случаях, когда линейный список реализован на массиве). Обозначая количество допустимых значений как CAR (кардинальное число), получим формулу $CAR() = CAR(BaseType)^0 + CAR(BaseType)^1 + \ldots + CAR(BaseType)^max$.

Задание 1.2.5:

Для последовательных линейных списков, использующих для хранения элементов массивы, доступ к элементам является прямым — при заданном индексе элемента, его адрес находится за фиксированное, не связанное с этим индексом число операций. Для связных линейных списков, использующих «записи» и систему указателей, связывающую соседние элементы, доступ к элементам является последовательным: необходимо идти от первого элемента к тому, с которым он связан, и так до тех пор, пока не будет достигнут требуемый.

Логический уровень:

СД типа «линейный список» не является встроенной, и потому описать его на логическом уровне (представить на языке программирования) можно только для определенной реализации. Приведем здесь описание для той реализации, которая будет выполнена в задании 2 этой лабораторной работы — то есть для односвязного линейного списка на массиве.

```
1  #include <stdio.h>
2
3  typedef int BaseType;
4
5  typedef unsigned ptrel;
6
7  typedef struct {
8      BaseType data;
9      ptrel next;
10 } element;
11
12 typedef struct {
13     ptrel start;
14     ptrel ptr;
15     unsigned int N;
16 } List;
17
18 void PrintList(List *L, element *cont) {
19     L->ptr = L->start;
```

```c
    for (int i = 0; i < L->N; i++) {
        printf("%d, ", cont[L->ptr].data);
        L->ptr = cont[L->ptr].next;
    }
    printf("\b\b \n");
}

int main() {
    element ListsContainer[10] = {
            {0, 2},
            {1, 3},
            {0, 6},
            {5, 5},
            {-7, 0},
            {6, 4},
            {0, 9},
            {2, 0},
            {3, 7},
            {0, 0}
    };
    List L1, L2;
    L1 = (List) {1, 1, 4};
    L2 = (List) {8, 8, 2};
    PrintList(&L1, ListsContainer);
    PrintList(&L2, ListsContainer);
}
```

../АСД 5 си/example.c

Увы, из-за отсутствия функций, облегчающих заполнение массива записями и вообще изменение полей списка (они будут реализованы в задании 2), она выглядит несколько громоздко. По сути здесь мы объявляем тип базового элемента, создаем структуру-«запись», хранящую значение базового типа и индекс (в массиве) элемента, идущего следующим в текущем списке, а потом создаем структуру-«список», хранящую указатель на первый и текущий элементы и количество элементов в списке на данный момент. В заданном массиве ListsContainer хранятся два списка с элементами {1, 5, 6, -7} и {3, 2}. Начало первого списка — элемент под индексом 1, второго — элемент под индексом 8. Идя по элементам с индексами в полях next, начиная от стартового элемента, рано или поздно мы придем к элементу под индексом 0 — началу списка свободных элементов. Увидеть, что элементы списков действительно связаны, помогает функция PrintList (мы можем увидеть результаты вывода на скриншоте ниже).

```
"C:\Users\sovac\Desktop\АСД\АСД 5 си\example.exe"
1, 5, 6, -7
3, 2

Process finished with exit code 0
```

Индивидуальное задание; вариант 21

Модуль 5: Элементы ОЛС находятся в массиве *Memlist*, расположенном в статической памяти. Базовый тип зависит от задачи. «Свободные» элементы массива объединяются в список, на начало которого указывает поле-указатель первого элемента массива. Выделение памяти под информационную часть элемента ОЛС и запись в нее значения происходит при выполнении процедуры *PutList*. При выполнении процедуры *GetList* память, занимаемая элементом, освобождается. **Реализация на языке С:**

```
#if !defined(__LIST5_H)
#define SizeList 100 const short ListOk = 0;
const short ListNotMem = 1;
const short ListUnder = 2;
const short ListEnd = 3;
typedef <определить> BaseType;
typedef unsigned ptrel;
typedef struct element {basetype data; ptrel next; };
typedef struct List {ptrel Start; ptrel ptr; unsigned int N}
element MemList[SizeList];
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 FullList(List *L)
int EndList(List *L)
usigned 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)
void InitMem()/*присваивает Flag каждого элемента в 0*/
int EmptyMem() /*возвращает 1, если в массиве нет свободных элементов*/
unsigned NewMem()//возвращает номер свободного элемента
void DisposeMem(unsigned n) /*делает n-й элемент массива свободным*/
#endif
```

Задача 10: Проверить, удовлетворяют ли элементы списка (базовый тип *integer*) закону

$x = f(_0, h)$, где — элемент списка, h — шаг, x_0 — начальный элемент списка. Пример: $x_0 = 5$, $h = 1$. $x_1 = 6$, $x_2 = 7$, $x_3 = 8$... Элементы списка удовлетворяют закону $= f(5, 1)$.

Задание 2:

Разделим код выполнения этого задания 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: BaseType — тип, элементы которого хранит список (в нашем случае это целые числа int); ptrel — беззнаковое целое число, индекс элемента массива MemList, где хранится некоторый элемент; element — запись, состоящая из значения элемента и индекса, по которому хранится следующий элемент; List — структура, хранящая индексы начального и текущего элемента, а также их общее количество. Только после этого будут объявлены прототипы функций для работы со списками.

```
#if !defined(__LIST5_H)
#define __LIST5_H

#define SizeList 100

const short ListOk = 0;
const short ListNotMem = 1;
const short ListUnder = 2;
const short ListEnd = 3;
short ListError;

typedef int BaseType;

typedef unsigned ptrel;

typedef struct {
    BaseType data;
    ptrel next;
} element;

typedef struct {
    ptrel start;
    ptrel ptr;
    unsigned int N;
} List;

element MemList[SizeList];

```

```
29 //Инициализация списка, который создается по адресу L
30 void InitList(List *L);
31 //Включение элемента со значением E в список по адресу L
32 void PutList(List *L, BaseType E);
33 //Исключение элемента из списка по адресу L и сохранение его в
      переменной по адресу E
34 void GetList(List *L, BaseType *E);
35 //Чтение элемента списка по адресу L и сохранение его в переменной по
      адресу E
36 void ReadList(List *L,BaseType *E);
37 //Возвращает 1, если список по адресу L не пуст, И 0 в противном случае
38 int FullList(List *L);
39 // проверка: является ли элемент списка по адресу L последним
40 int EndList(List *L);
41 //Возвращает количество элементов в списке по адресу L
42 unsigned int Count(List *L);
43 //Установка в начало списка по адресу L
44 void BeginPtr(List *L);
45 //Установка в конец списка по адресу L
46 void EndPtr(List *L);
47 //Переход к следующему элементу в списке по адресу L
48 void MovePtr(List *L);
49 //Переход к nму- элементу в списке по адресу L
50 void MoveTo(List *L, unsigned int n);
51 //Удаление списка по адресу L
52 void DoneList(List *L);
53 //Копирование списка по адресу L1 в список по адресу L2
54 void CopyList(List *L1,List *L2);
55 //Связывает все элементы массива в список свободных элементов
56 void InitMem();
57 //Возвращает 1, если в массиве нет свободных элементов, и 0 в
      противном случае
58 int EmptyMem();
59 //Возвращает номер свободного элемента и исключает его из ССЭ
60 unsigned NewMem();
61 //делает nй- элемент массива свободным и включает его в ССЭ
62 void DisposeMem(unsigned n);
63
64 #endif
```

../АСД 5 си/List.h

Реализация функций будет вынесена в отдельный файл со следующим содержанием.

Примечание: функция PutList подразумевает, что элемент вставляется после элемента, на который указывает рабочий указатель, поэтому, чтобы вставить элемент в самое начало списка, рабочий указатель должен быть равен нулю, при условии, что нулю не равен стартовый указатель (если же стартовый указатель равен 0, это запускает отдельный алгоритм вставки в пустой список).

```
#if !defined(__LIST5_C)
#define __LIST5_C
#include "List.h"

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


void InitList(List *L) {
     L->start = 0;
     L->ptr = 0;
     L->N = 0;
}

void PutList(List *L, BaseType E) {
    if (EmptyMem()) {
        ListError = ListNotMem;
        exit(ListError);
    } else {
        ptrel pntr = NewMem();
        MemList[pntr].data = E;
        if (L->start == 0) {
            MemList[pntr].next = 0;
            L->start = pntr;
        } else if (L->ptr == 0) {
            MemList[pntr].next = L->start;
            L->start = pntr;
        } else {
            MemList[pntr].next = MemList[L->ptr].next;
            MemList[L->ptr].next = pntr;
        }
        L->ptr = pntr;
        L->N++;
        ListError = ListOk;
    }
}

void GetList(List *L, BaseType *E)  {
    if (L->start == 0) {
        ListError = ListUnder;
        exit(ListError);
    } else if (L->ptr == 0) {
        ListError = ListEnd;
        exit(ListError);
    } else {
        *E = MemList[L->ptr].data;
        ptrel pntr = L->ptr;
```

```
48          if (pntr == L->start) {
49              L->start = MemList[pntr].next;
50          } else {
51              BeginPtr(L);
52              while (MemList[L->ptr].next != pntr) {
53                  MovePtr(L);
54              }
55              MemList[L->ptr].next = MemList[pntr].next;
56          }
57          L->ptr = MemList[pntr].next;
58          DisposeMem(pntr);
59          L->N--;
60          ListError = ListOk;
61      }
62  }
63
64  void ReadList(List *L,BaseType *E) {
65      if (L->ptr == 0) {
66          ListError = ListUnder;
67          exit(ListError);
68      }
69      *E = MemList[L->ptr].data;
70      ListError = ListOk;
71  }
72
73  int FullList(List *L) {
74      return L->start != 0;
75  }
76
77  int EndList(List *L) {
78      if (L->ptr == 0) {
79          ListError = ListEnd;
80          exit(ListError);
81      }
82      ListError = ListOk;
83      return MemList[L->ptr].next == 0;
84  }
85
86  unsigned int Count(List *L) {
87      return L->N;
88  }
89
90  void BeginPtr(List *L) {
91      L->ptr = L->start;
92  }
93
94  void EndPtr(List *L) {
```

```
95      BeginPtr(L);
96      for (int i = 1; i < L->N; i++) {
97          MovePtr(L);
98      }
99  }
100
101 void MovePtr(List *L) {
102     if (L->ptr == 0) {
103         ListError = ListEnd;
104         exit(ListError);
105     }
106     L->ptr = MemList[L->ptr].next;
107     ListError = ListOk;
108 }
109
110 void MoveTo(List *L, unsigned int n) {
111     BeginPtr(L);
112     if (n > L->N) {
113         ListError = ListEnd;
114         exit (ListError);
115     }
116     for (int i = 0; i < n; i++) {
117         MovePtr(L);
118     }
119     ListError = ListOk;
120 }
121
122 void DoneList(List *L) {
123     BeginPtr(L);
124     if (FullList(L)) {
125         while ((MemList[L->ptr].next) != 0) {
126             BaseType trash;
127             GetList(L, &trash);
128         }
129     }
130     free(L);
131 }
132
133 void CopyList(List *L1,List *L2) {
134     ptrel ptr_value = L1->ptr;
135     BeginPtr(L1);
136     BeginPtr(L2);
137     while (L2->start != 0) {
138         BaseType trash;
139         GetList(L2, &trash);
140     }
141     for (int i = 0; i < L1->N; i++) {
```

```
142         BaseType cur_data;
143         ReadList(L1, &cur_data);
144         PutList(L2, cur_data);
145         MovePtr(L1);
146     }
147     L1->ptr = ptr_value;
148     ListError = ListOk;
149 }
150 
151 void InitMem() {
152     for (int i = 0; i < SizeList-1; i++) {
153         MemList[i].next = i+1;
154     }
155     MemList[SizeList-1].next = 0;
156 }
157 
158 int EmptyMem() {
159     return MemList[0].next == 0;
160 }
161 
162 unsigned NewMem() {
163     ptrel first_free = MemList[0].next;
164     MemList[0].next = MemList[first_free].next;
165     return first_free;
166 }
167 
168 void DisposeMem(unsigned n) {
169     ptrel first_free = MemList[0].next;
170     MemList[0].next = n;
171     MemList[n].next = first_free;
172 }
173 
174 #endif
```

../АСД 5 си/List.c

Для нормальных (не вызывающих ошибки и аварийного завершения) сценариев большинства функций (за исключением тех, что работают не со списками, а непосредственно с массивом MemList, на основе которого они хранятся, а также уничтожающей список функции DoneList), 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.

```
1 #include "List.h"
2 #include "List.c"
3 
4 #include <stdio.h>
5 #include <assert.h>
6 
```

```
7
8  void Test_InitList() {
9      List L;
10     InitList(&L);
11     assert(L.start == 0 && L.ptr == 0 && L.N == 0);
12 }
13
14 void Test_PutList_InEmpty() {
15     List L;
16     InitList(&L);
17     PutList(&L, 5);
18     assert(MemList[L.start].data == 5 && MemList[L.ptr].data == 5 &&
       MemList[L.ptr].next == 0 &&L.N == 1);
19 }
20 void Test_PutList_AtEnd() {
21     List L;
22     InitList(&L);
23     PutList(&L, 5);
24     PutList(&L, 7);
25     assert(MemList[L.start].data == 5 && MemList[L.ptr].data == 7 &&
       MemList[MemList[L.start].next].data == 7);
26     assert(MemList[L.start].next == L.ptr && MemList[L.ptr].next == 0
       && L.N == 2);
27 }
28 void Test_PutList_AtBeginning() {
29     List L;
30     InitList(&L);
31     PutList(&L, 5);
32     L.ptr = 0;
33     PutList(&L, 7);
34     assert(MemList[L.start].data == 7 && MemList[L.ptr].data == 7 &&
       MemList[MemList[L.start].next].data == 5);
35     assert(L.start == L.ptr && MemList[MemList[L.ptr].next].next == 0
       && L.N == 2);
36 }
37 void Test_PutList_Between() {
38     List L;
39     InitList(&L);
40     PutList(&L, 5);
41     ptrel ptr_at_first = L.ptr;
42     PutList(&L, 7);
43     L.ptr = ptr_at_first;
44     PutList(&L, 4);
45     assert(MemList[L.start].data == 5 && MemList[L.ptr].data == 4 &&
       MemList[MemList[L.start].next].data == 4);
46     assert(MemList[MemList[L.ptr].next].data == 7 &&
       MemList[MemList[L.ptr].next].next == 0 && L.N == 3);
```

```
}
void Test_PutList() {
    Test_PutList_InEmpty();
    Test_PutList_AtEnd();
    Test_PutList_AtBeginning();
    Test_PutList_Between();
}

void Test_GetList_OneAndOnly() {
    List L;
    InitList(&L);
    PutList(&L, 5);
    BaseType container;
    GetList(&L, &container);
    assert(container == 5 && L.start == 0 && L.ptr == 0 && L.N == 0);
}
void Test_GetList_AtEnd() {
    List L;
    InitList(&L);
    PutList(&L, 5);
    PutList(&L, 7);
    BaseType container;
    GetList(&L, &container);
    assert(container == 7 && MemList[L.start].data == 5);
    assert(L.ptr == 0 && MemList[L.start].next == 0 && L.N == 1);
}
void Test_GetList_AtBeginning() {
    List L;
    InitList(&L);
    PutList(&L, 5);
    ptrel ptr_at_first = L.ptr;
    PutList(&L, 7);
    L.ptr = ptr_at_first;
    BaseType container;
    GetList(&L, &container);
    assert(container == 5 && MemList[L.start].data == 7 &&
   MemList[L.ptr].data == 7);
    assert(MemList[L.start].next == 0 && MemList[L.ptr].next == 0 &&
   L.N == 1);
}
void Test_GetList_Between() {
    List L;
    InitList(&L);
    PutList(&L, 5);
    PutList(&L, 7);
    ptrel ptr_at_second = L.ptr;
    PutList(&L, 4);
```

```
92      L.ptr = ptr_at_second;
93      BaseType container;
94      GetList(&L, &container);
95      assert(container == 7 && MemList[L.start].data == 5 &&
        MemList[L.ptr].data == 4 && MemList[MemList[L.start].next].data ==
        4);
96      assert(MemList[L.ptr].next == 0 && L.N == 2);
97  }
98  void Test_GetList() {
99      Test_GetList_OneAndOnly();
100     Test_GetList_AtEnd();
101     Test_GetList_AtBeginning();
102     Test_GetList_Between();
103 }
104
105 void Test_ReadList_OneAndOnly() {
106     List L;
107     PutList(&L, 5);
108     BaseType container;
109     ReadList(&L, &container);
110     assert(container == 5 && MemList[L.ptr].data == 5);
111 }
112 void Test_ReadList_AtEnd() {
113     List L;
114     PutList(&L, 5);
115     PutList(&L, 7);
116     BaseType container;
117     ReadList(&L, &container);
118     assert(container == 7 && MemList[L.ptr].data == 7);
119 }
120 void Test_ReadList_AtBeginning() {
121     List L;
122     PutList(&L, 5);
123     ptrel ptr_at_first = L.ptr;
124     PutList(&L, 7);
125     L.ptr = ptr_at_first;
126     BaseType container;
127     ReadList(&L, &container);
128     assert(container == 5 && MemList[L.ptr].data == 5);
129 }
130 void Test_ReadList_Between() {
131     List L;
132     PutList(&L, 5);
133     PutList(&L, 7);
134     ptrel ptr_at_second = L.ptr;
135     PutList(&L, 4);
136     L.ptr = ptr_at_second;
```

```
137     BaseType container;
138     ReadList(&L, &container);
139     assert(container == 7 && MemList[L.ptr].data == 7);
140 }
141 void Test_ReadList() {
142     Test_ReadList_OneAndOnly();
143     Test_ReadList_AtEnd();
144     Test_ReadList_AtBeginning();
145     Test_ReadList_Between();
146 }
147 
148 void Test_FullList_Empty() {
149     List L;
150     InitList(&L);
151     assert(!FullList(&L));
152 }
153 void Test_FullList_SomeElementsIn() {
154     List L;
155     InitList(&L);
156     PutList(&L, 5);
157     PutList(&L, 7);
158     PutList(&L, 4);
159     assert(FullList(&L));
160 }
161 void Test_FullList_PtrAt0() {
162     List L;
163     InitList(&L);
164     PutList(&L, 5);
165     PutList(&L, 7);
166     PutList(&L, 4);
167     L.ptr = 0;
168     assert(FullList(&L));
169 }
170 void Test_FullList() {
171     Test_FullList_Empty();
172     Test_FullList_SomeElementsIn();
173     Test_FullList_PtrAt0();
174 }
175 
176 void Test_EndList_OneElement() {
177     List L;
178     InitList(&L);
179     PutList(&L, 5);
180     assert(EndList(&L));
181 }
182 void Test_EndList_PtrAtEnd() {
183     List L;
```

```
184     InitList(&L);
185     PutList(&L, 5);
186     PutList(&L, 7);
187     PutList(&L, 4);
188     assert(EndList(&L));
189 }
190 void Test_EndList_PtrInMiddle() {
191     List L;
192     InitList(&L);
193     PutList(&L, 5);
194     PutList(&L, 7);
195     ptrel ptr_at_second = L.ptr;
196     PutList(&L, 4);
197     L.ptr = ptr_at_second;
198     assert(!EndList(&L));
199 }
200 
201 void Test_EndList() {
202     Test_EndList_OneElement();
203     Test_EndList_PtrAtEnd();
204     Test_EndList_PtrInMiddle();
205 }
206 
207 void Test_Count_Empty() {
208     List L;
209     InitList(&L);
210     assert(Count(&L) == 0);
211 }
212 void Test_Count_OneElement() {
213     List L;
214     InitList(&L);
215     PutList(&L, 5);
216     assert(Count(&L) == 1);
217 }
218 void Test_Count_SomeElements() {
219     List L;
220     InitList(&L);
221     PutList(&L, 5);
222     PutList(&L, 7);
223     PutList(&L, 4);
224     assert(Count(&L) == 3);
225 }
226 void Test_Count() {
227     Test_Count_Empty();
228     Test_Count_OneElement();
229     Test_Count_SomeElements();
230 }
```

```
231
232 void Test_BeginPtr_Empty() {
233     List L;
234     InitList(&L);
235     BeginPtr(&L);
236     assert(L.ptr == 0);
237 }
238 void Test_BeginPtr_OneElement() {
239     List L;
240     InitList(&L);
241     PutList(&L, 5);
242     ptrel ptr_start = L.ptr;
243     BeginPtr(&L);
244     assert(L.ptr == ptr_start);
245 }
246 void Test_BeginPtr_SomeElements() {
247     List L;
248     InitList(&L);
249     PutList(&L, 5);
250     ptrel ptr_start = L.ptr;
251     PutList(&L, 7);
252     PutList(&L, 4);
253     BeginPtr(&L);
254     assert(L.ptr == ptr_start);
255 }
256 void Test_BeginPtr() {
257     Test_BeginPtr_Empty();
258     Test_BeginPtr_OneElement();
259     Test_BeginPtr_SomeElements();
260 }
261
262 void Test_EndPtr_Empty() {
263     List L;
264     InitList(&L);
265     BeginPtr(&L);
266     assert(L.ptr == 0);
267 }
268 void Test_EndPtr_OneElement() {
269     List L;
270     InitList(&L);
271     PutList(&L, 5);
272     ptrel ptr_end = L.ptr;
273     EndPtr(&L);
274     assert(L.ptr == ptr_end);
275 }
276 void Test_EndPtr_SomeElements() {
277     List L;
```

```
278     InitList(&L);
279     PutList(&L, 5);
280     PutList(&L, 7);
281     PutList(&L, 4);
282     ptrel ptr_end = L.ptr;
283     BeginPtr(&L);
284     EndPtr(&L);
285     assert(L.ptr == ptr_end);
286 }
287 void Test_EndPtr() {
288     Test_EndPtr_Empty();
289     Test_EndPtr_OneElement();
290     Test_EndPtr_SomeElements();
291 }
292 
293 void Test_MovePtr_OneElementTo0() {
294     List L;
295     InitList(&L);
296     PutList(&L, 5);
297     MovePtr(&L);
298     assert(L.ptr == 0);
299 }
300 void Test_MovePtr_SomeElementsTo0() {
301     List L;
302     InitList(&L);
303     PutList(&L, 5);
304     PutList(&L, 7);
305     PutList(&L, 4);
306     MovePtr(&L);
307     assert(L.ptr == 0);
308 }
309 void Test_MovePtr_SomeElementsFromFirst() {
310     List L;
311     InitList(&L);
312     PutList(&L, 5);
313     PutList(&L, 7);
314     ptrel ptr_second = L.ptr;
315     PutList(&L, 4);
316     BeginPtr(&L);
317     MovePtr(&L);
318     assert(L.ptr == ptr_second);
319 }
320 void Test_MovePtr_SomeElementsToLast() {
321     List L;
322     InitList(&L);
323     PutList(&L, 5);
324     PutList(&L, 7);
```

```
    PutList(&L, 4);
    ptrel ptr_last = L.ptr;
    BeginPtr(&L);
    MovePtr(&L);
    MovePtr(&L);
    assert(L.ptr == ptr_last);
}
void Test_MovePtr() {
    Test_MovePtr_OneElementTo0();
    Test_MovePtr_SomeElementsTo0();
    Test_MovePtr_SomeElementsFromFirst();
    Test_MovePtr_SomeElementsToLast();
}

void Test_MoveTo_OneElement() {
    List L;
    InitList(&L);
    PutList(&L, 5);
    ptrel ptr_first = L.ptr;
    L.ptr = 0;
    MoveTo(&L, 0);
    assert(L.ptr == ptr_first);
}
void Test_MoveTo_SomeElementsToFirst() {
    List L;
    InitList(&L);
    PutList(&L, 5);
    ptrel ptr_first = L.ptr;
    PutList(&L, 7);
    PutList(&L, 4);
    MoveTo(&L, 0);
    assert(L.ptr == ptr_first);
}
void Test_MoveTo_SomeElementsToMiddle() {
    List L;
    InitList(&L);
    PutList(&L, 5);
    PutList(&L, 7);
    ptrel ptr_second = L.ptr;
    PutList(&L, 4);
    MoveTo(&L, 1);
    assert(L.ptr == ptr_second);
}
void Test_MoveTo_SomeElementsToLast() {
    List L;
    InitList(&L);
    PutList(&L, 5);
```

```
372     PutList(&L, 7);
373     PutList(&L, 4);
374     ptrel ptr_last = L.ptr;
375     BeginPtr(&L);
376     MoveTo(&L, 2);
377     assert(L.ptr == ptr_last);
378 }
379 void Test_MoveTo() {
380     Test_MoveTo_OneElement();
381     Test_MoveTo_SomeElementsToFirst();
382     Test_MoveTo_SomeElementsToMiddle();
383     Test_MoveTo_SomeElementsToLast();
384 }
385
386 void Test_CopyList_Empty() {
387     List L1, L2;
388     InitList(&L1);
389     InitList(&L2);
390     CopyList(&L1, &L2);
391     assert(L2.start == 0 && L2.ptr == 0 && L2.N == 0);
392 }
393 void Test_CopyList_ToEmpty() {
394     List L1, L2;
395     InitList(&L1);
396     PutList(&L1, 5);
397     PutList(&L1, 7);
398     PutList(&L1, 4);
399     InitList(&L2);
400     CopyList(&L1, &L2);
401     assert(L2.N == 3);
402     BeginPtr(&L1);
403     BeginPtr(&L2);
404     for (int i = 0; i < L1.N; i++) {
405         BaseType cont1, cont2;
406         ReadList(&L1, &cont1);
407         ReadList(&L2, &cont2);
408         assert(cont1 == cont2);
409         MovePtr(&L1);
410         MovePtr(&L2);
411     }
412     assert(L2.ptr == 0);
413 }
414 void Test_CopyList_ToFull() {
415     InitMem();
416     List L1, L2;
417     InitList(&L1);
418     PutList(&L1, 5);
```

```
419     PutList(&L1, 7);
420     PutList(&L1, 4);
421     InitList(&L2);
422     PutList(&L2, 2);
423     PutList(&L2, -6);
424     PutList(&L2, 1);
425     PutList(&L2, 8);
426     CopyList(&L1, &L2);
427     assert(L2.N == 3);
428     BeginPtr(&L1);
429     BeginPtr(&L2);
430     for (int i = 0; i < L1.N; i++) {
431         BaseType cont1, cont2;
432         ReadList(&L1, &cont1);
433         ReadList(&L2, &cont2);
434         assert(cont1 == cont2);
435         MovePtr(&L1);
436         MovePtr(&L2);
437     }
438     assert(L2.ptr == 0);
439 }
440
441 void Test_CopyList() {
442     Test_CopyList_Empty();
443     Test_CopyList_ToEmpty();
444     Test_CopyList_ToFull();
445 }
446 void List_Test() {
447     Test_InitList();
448     Test_PutList();
449     Test_GetList();
450     Test_ReadList();
451     Test_FullList();
452     Test_EndList();
453     Test_Count();
454     Test_BeginPtr();
455     Test_EndPtr();
456     Test_MovePtr();
457     Test_MoveTo();
458     Test_CopyList();
459 }
460
461 int main() {
462     InitMem();
463     List_Test();
464     printf("All is OK!");
```

```
465 }
```

../АСД 5 си/List_test.c

Запустив программу, можем самостоятельно убедиться, что все тесты прошли успешно:

```
"C:\Users\sovac\Desktop\АСД\АСД 5 си\List_test.exe"
All is OK!
Process finished with exit code 0
```

Задание 3:

Теперь, когда все функции, обязательные для СД типа «линейный список», реализованы, можем решить задачу, в ходе которой нам предстоит работать с линейными списками. Прототип функции, решающей задачу добавим в заголовочный файл, затем вставим основной код в файл реализации, а в файле с автоматизированными тестами рассмотрим несколько сценариев: список содержит один элемент (будем считать, что тогда он удовлетворяет условию); список содержит несколько элементов, и шаг между ними одинаков и соответствует заданному; список содержит несколько элементов, но на одном из них шаг отклоняется от заданного, и часть последовательности после него перестает соответствовать условию.

Примечание: случай, когда последовательность пуста, будем считать ошибкой: программа аварийно завершает выполнение с кодом ListUnder.

```
1  int IsProgression (List *P, int h) {
2      BeginPtr(P);
3      int x0;
4      ReadList(P, &x0);
5      for (int i = 1; i < P->N; i++) {
6          int x;
7          MovePtr(P);
8          ReadList(P, &x);
9          if (x != x0 + h*i) {
10             return 0;
11         }
12     }
13     ListError = ListOk;
14     return 1;
15 }
```

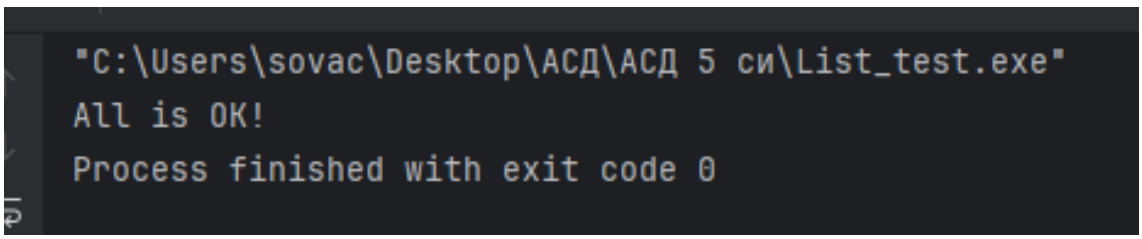
../АСД 5 си/List.c

```
1  void Test_IsProgression_OneElement() {
2      List P;
3      InitList(&P);
4      PutList(&P, 6);
```

```
5       assert(IsProgression(&P, 1));
6   }
7   void Test_IsProgression_SomeElementsRow() {
8       List P;
9       InitList(&P);
10      PutList(&P, 6);
11      PutList(&P, 9);
12      PutList(&P, 12);
13      PutList(&P, 15);
14      assert(IsProgression(&P, 3));
15  }
16  void Test_IsProgression_SomeElementsNotRow() {
17      List P;
18      InitList(&P);
19      PutList(&P, 6);
20      PutList(&P, 9);
21      PutList(&P, 11);
22      PutList(&P, 14);
23      assert(!IsProgression(&P, 3));
24  }
25  void Test_IsProgression() {
26      Test_IsProgression_OneElement();
27      Test_IsProgression_SomeElementsRow();
28      Test_IsProgression_SomeElementsNotRow();
29  }
30
31  void List_Test() {
32      /*...*/
33      Test_IsProgression();
34  }
35
36  int main() {
37      InitMem();
38      List_Test();
39      printf("All is OK!");
40  }
```

../АСД 5 си/List_test.c

Запустив программу, можем убедиться, что и этот тест полностью пройден:



Вывод:

В ходе лабораторной работы дали характеристику СД типа «линейный список» и ее подвидам (ПЛС, ОЛС, ДЛС), форматам их представления, реализовали один из них (односвязный линейный список на основе массива в статической памяти), написали ряд базовых функций для работы с односвязными линейными списками в этом формате.