Лабораторная работа №6.

ЛИНЕЙНЫЕ СВЯЗНЫЕ СПИСКИ

1. Цель работы

Освоение методов обработки линейных списковых структур, приобретение навыков составления и отладки соответствующих программ.

2. Основные сведения

Рассмотрим организацию системы учета вкладов в отделении сбербанка. Будем считать, что в простейшем случае информация о вкладе содержит фамилию вкладчика и сумму вклада в рублях:

```
struct ВКЛАД {
      char  ФАМ [20];
      long int  СУММА;
}
```

Пусть необходимо размещать вклады в алфавитном порядке фамилий вкладчиков. Операции, которые должна выполнять система: 1) определить, имеется ли вкладчик с данной фамилией (НАЙТИ), 2) включить новый вклад (ВКЛЮЧИТЬ), 3) вычеркнуть вклад из системы (ИСКЛЮЧИТЬ).

Для обработки данных о всех вкладах можно использовать массив вкладов: **ВКЛАД** ВСЕ [1000], тогда поиск вклада может быть выполнен быстро, например, при числе вкладов К при использовании метода деления пополам: максимальное время поиска примерно пропорционально целой части двоичного логарифма К: $\lceil \log_2 K \rceil$.

Однако операция ВКЛЮЧИТЬ в общем случае требует сдвига части массива. Удаление вклада приводит либо к незаполненным данными "окнам" (когда вкладчик отмечается как отсутствующий: например, поле ФАМ элемента массива ВСЕ заполняется пробелами или пустой строкой), либо опять требует сдвига части массива. Сдвиг массива – это нежелательное действие, которое занимает время, пропорциональное длине массива К. Кроме того, неиспользуемые компоненты массива занимают память.

Разрешить возникшую проблему выполнения операций ВКЛЮЧИТЬ и ИСКЛЮЧИТЬ можно в результате отказа от статического размещения элементов массива-вкладов и организации линейного связного списка в динамической памяти.

Список состоит из элементов, каждый из которых в общем случае является структурой, в которой выделены содержательная и вспомогательная части. Вспомогательная часть используется для организации связей между элементами списка и содержит одно или несколько полей типа указатель.

В системе учета вкладов, использующей динамические списки, содержательная часть элементов списка остается такой же, как указано выше, и к ней добавляется в простейшем случае одно поле для указания следующего элемента списка СЛЕД. Поле СЛЕД должно быть переменной типа указатель, который можно назвать СВЯЗЬ:

```
struct ВКЛАД {
      char ФАМ  [20];
      long int СУММА;
      ВКЛАД * СЛЕД;
   }  // ВКЛАД
typedef ВКЛАД * СВЯЗЬ;
```

Тогда информацию, например о четырех вкладах, можно представить в виде списка (рис. 1). Переменная Fst типа ВКЛАД * указывает на первый элемент списка, а поле СЛЕД в последнем элементе имеет значение **nullptr**.

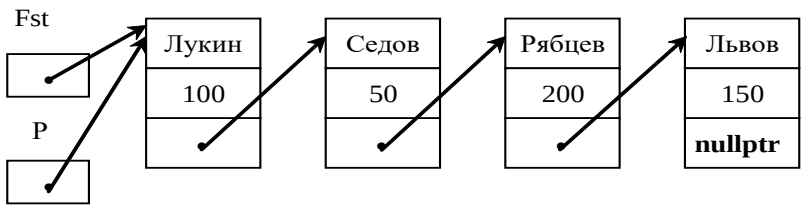

Рис. 1. Список вкладчиков

Сформировать список с информацией о вкладах (рис. 1) можно с помощью программы, включающую следующую функцию:

```
typedef ВКЛАД * СВЯЗЬ;
void CreateList(СВЯЗЬ Fst) {
  СВЯЗЬ P;
  char S[20];

  Fst=nullptr;
  cout << "Фамилия: "; cin >> S;
  while (strcmp(S, "#") != 0) {  // Выход по #
    P = new ВКЛАД;
    P->СЛЕД = Fst;
    strcpy(P->ФАМ, S);
    cout >> "Сумма:   ";  cin >> P->СУММА;
    Fst = P;
    cout << "Фамилия: "; cin >> S;
   };
}
```

Обратите внимание на способ выделения пространства для стольких структур. Инструментом для этого служит операция new. С ее помощью можно создавать динамические структуры. «Динамические» - означает выделение памяти во время выполнения программы, а не во время её компиляции.

Применение new со структурами состоит из двух частей: создание структуры и обращение к ее членам.

Для создания структуры вместе с операцией new указывается тип структуры:

```
СВЯЗЬ P; // то же что и ВКЛАД * P;
P = new ВКЛАД;
```

Здесь создаётся безымянная структура типа ВКЛАД и указателю P присваивается адрес выделенного участка памяти достаточного размера, чтобы вместить тип ВКЛАД. Обратите внимание, что синтаксис в точности такой же, как и для встроенных типов C++.

Более сложная часть – доступ к членам. Когда вы создаете динамическую структуру, то не можете применить *операцию принадлежности, операцию членства* (операция точка) к имени структуры, поскольку эта структура безымянна. Все, что у вас есть – ее адрес. В C++ предусмотрена специальная операция для этой ситуации – *операция косвенного членства, операция членства через указатель* (->). Эта операция, оформленная в виде тире с последующим значком «больше», означает для указателей на структуры то же самое, что операция точки для имен структур. Например, P->СУММА означает член СУММА структуры, на которую указывает P.

Существует еще один спосрб обращения к членам структур; он принимает во внимание, что если P – указатель на структуру, то *P – сама структура. Поэтому, если *P – структура, то

(*P).СУММА – ее член СУММА. Правила приоритетов операций C++ требуют применения скобок в этой конструкции (опретор принадлежности имеет более высокий приоритет, чем операция разыменовывания).

Последовательность выполнения первых шагов функции `CreateList` иллюстрирует рис. 2.

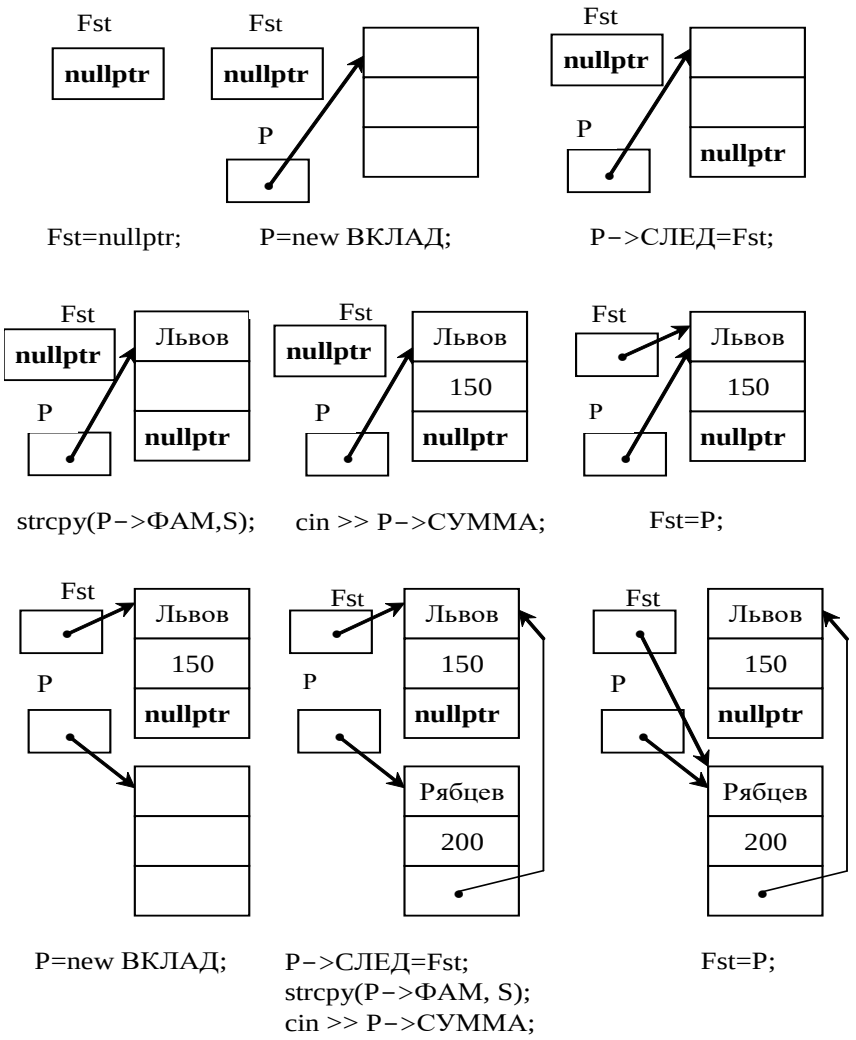

Рис. 2. Последовательность создания списка

Рассмотрим *поиск элемента* списка с заданным значением одного из полей. Пусть, например, необходимо увеличить сумму вклада вкладчика А. Для этого рассмотрим еще одну переменную типа СВЯЗЬ, назовем ее РТ, которая будет перемещаться по списку до обнаружения нужной фамилии:

```
PT = Fst;
while  (strcmp(PT->ФАМ, A) != 0)  PT = PT->СЛЕД;
```

Этот фрагмент можно пояснить так. Сначала РТ указывает на первый элемент списка.

Если фамилия вкладчика, на которую указывает переменная РТ (поле РТ->ФАМ) не совпадает с заданной фамилией А, нужно передвинуть РТ на следующий элемент списка, заданный полем СЛЕД записи, на который в данный момент выполнения указывает РТ: РТ->СЛЕД. Поиск завершается, когда фамилия вкладчика, на которую указывает РТ, совпадает с искомой фамилией А.

Процесс поиска вкладчика с фамилией Иванов среди вкладчиков банка показан на рис. 3.

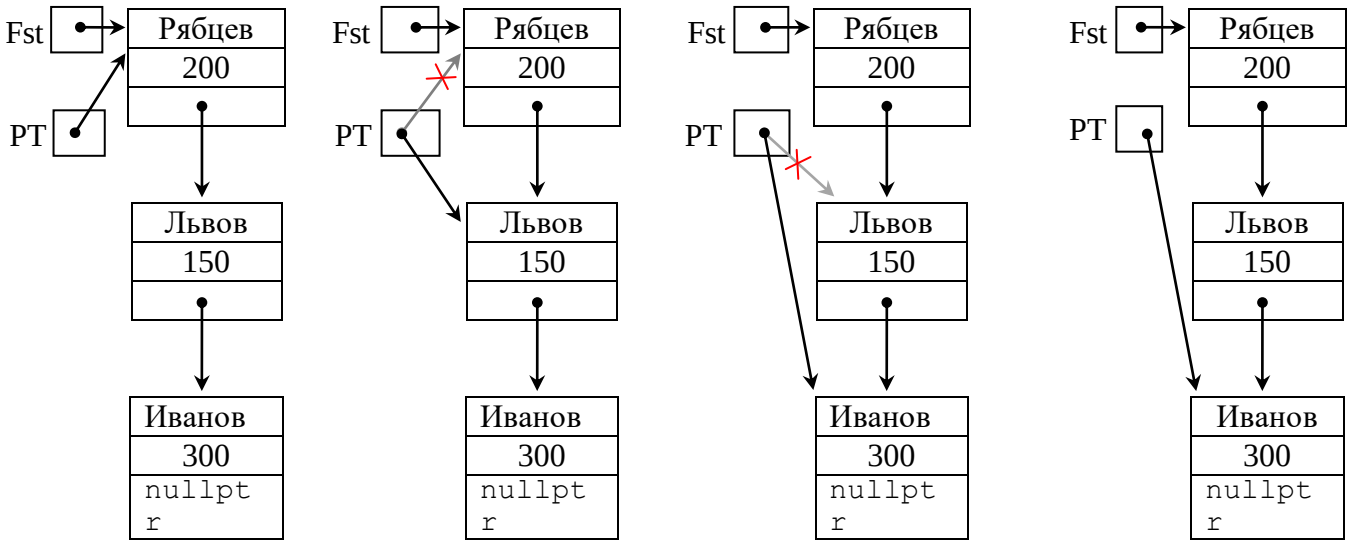

Рис. 3. Поиск элемента в списке

Приведённый фрагмент будет решать задачу ПОИСК только в том случае, когда можно гарантировать, что человек с фамилией А имеет в системе учёта некоторый вклад. Если это не так, то, "подойдя" к последнему элементу списка, переменная PT приобретает значение **nullptr**, и на следующем шаге выполнения цикла обращение к полю PT->ФАМ вызовет запрос к памяти по нулевому указателю и, возможно, будущее аварийное окончание программы, так как требуемого элемента просто не существует.

Опознать конец списка можно попытаться так:

```
PT = FST;
while ((PT != nullptr) and strcmp(PT->ФАМ, A)!=0)
   PT=PT->СЛЕД;  .
```

Это решение в общем случае не устранит аварийное окончание по той же самой причине: в конце списка PT=**nullptr** и элемента *PT с полем ФАМ, который требуется для выполнения операции сравнения, заключенной во вторую пару круглых скобок, в системе учета не создано.

Варианты правильного решения приведены ниже:

```
1) PT=FST;
   while ((PT != nullptr) and strcmp(PT->ФАМ, A)!=0)
      PT=PT->СЛЕД;
   if (strcmp(PT->ФАМ, A)!=0) PT= nullptr;
2) PT=FST; NotFound=true;  // Этот вариант лучше
   while ((PT != nullptr) and NotFound)
      if (strcmp(PT->ФАМ, A)==0)
         NotFound=false;   // То есть найдено
      else PT=PT->СЛЕД;
3)  PT=FST; NotFound=true;
   while ((PT != nullptr) and NotFound) {
      NotFound= strcmp(PT->ФАМ, A)!=0;
      if (NotFound)    // Более компактный вариант
         PT=PT->СЛЕД;
```

}

Во обоих случаях после выполнения фрагмента справедливо: если человек с фамилией A является вкладчиком, то PT указывает на его вклад, иначе PT=**nullptr**.

Время поиска элемента с заданным полем пропорционально длине списка O(n), так как при поиске в худшем случае просматривается последовательно весь список (см. рис. 3).

Следующая типовая задача состоит в том, чтобы *вставить новый элемент* в список. Для нашего примера это соответствует появлению нового вклада. Вставка возможна в начало списка (перед элементом, на который указывает переменная Fst) или после любого элемента, например, после элемента, на который указывает PT (рис. 4, 5):

```
НОВР = new ВКЛАД;
cin >> НОВР->ФАМ;  cin >> НОВР->СУММА; // рис. 4
НОВР->СЛЕД  =  PT->СЛЕД; // рис. 5,①
PT->СЛЕД  =  НОВР;    // рис 5,②
```

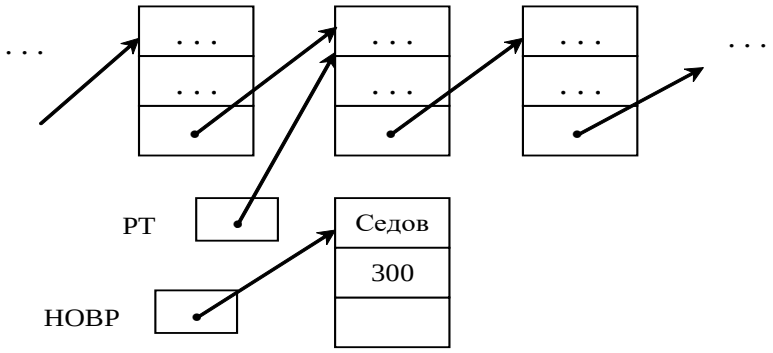

Рис. 4. Новый вклад перед включением

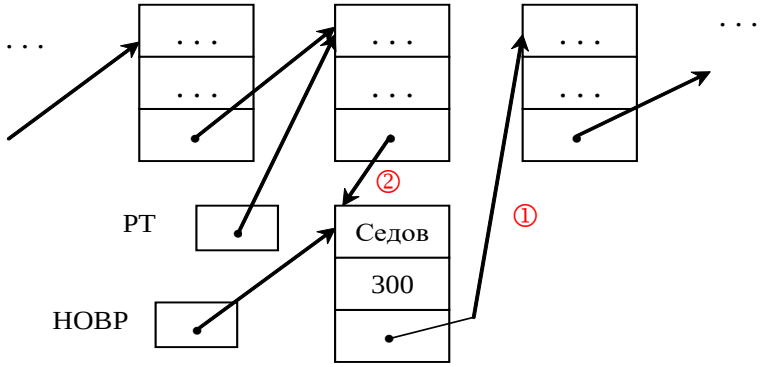

Рис. 5. Вклад после включения

Два решения еще одной задачи – задачи *удаления элемента* списка – показаны на рис. 6, 7. На рис. 6 удаляется элемент, следующий за элементом, на который указывает ссылочная переменная P. На рис. 7 на удаляемый элемент ссылается переменная P2.

В первом случае удаленный элемент недоступен в списке, но в память, отведенная для него оператором new, не освобождена и не доступна для формирования нового элемента. Во втором – после удаления элемента можно выполнить стандартную процедуру delete P2, которая возвратит занимаемую область памяти для дальнейшего использования. По этой причине второй вариант удаления предпочтительней первого. Можно также не уничтожать компонент, а добавить его в список свободных компонент, что уменьшит время выполнения программы.

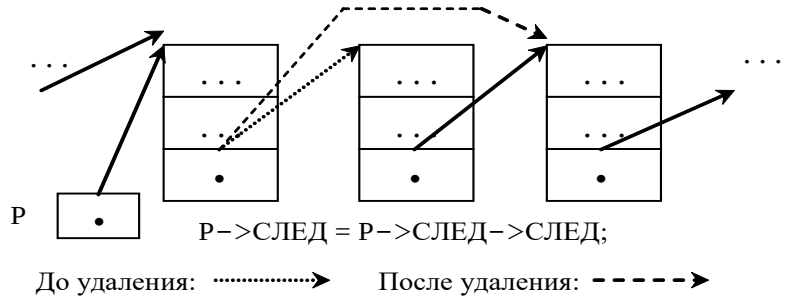

Рис. 6. Удаление элемента после указанного

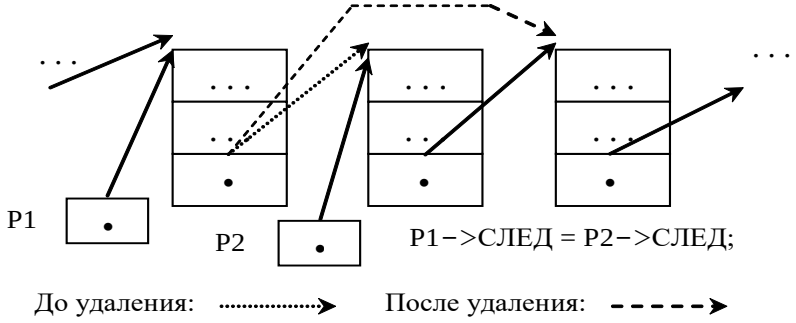

Рис. 7. Удаление указанного элемента

На рис. 8, 9 приведены примеры других видов списков. Элемент двусвязного линейного списка содержит два поля указателей, один из которых указывает на следующий элемент, а другой – на предыдущий. Кольцевой (циклический) список представляет «кольцо» элементов и является простой модификацией рассмотренного выше линейного списка: последний элемент в поле связи вместо значения nullptr содержит ссылку на первый элемент списка.

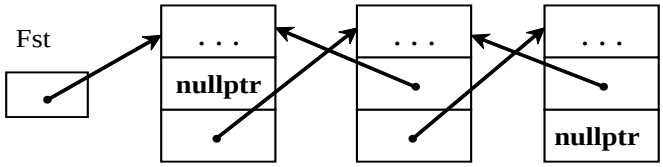

Рис. 8. Двусвязный список

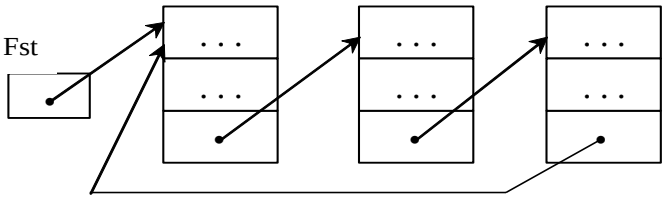

Рис. 9. Кольцевой односвязный список

Рассмотрим пример оформления действий со списками в виде функции. Приведем формирование двусвязного линейного списка. В «содержательной» части элементов списка записан один символ. Формирование списка заканчивается при вводе конца строки (клавиша «Enter»). На рис. 10 приведен пример результата работы функции FORM:

```
struct  ЭЛМНТ {
    char ИНФ;
    ЭЛМНТ * ЛЕВ;
    ЭЛМНТ * ПРАВ;
};
```

```
typdef
    ЭЛМНТ * СВ;

void FORM (СВ &ПЕРВ) { // или   ЭЛМНТ * &ПЕРВ
  СВ ТЕК;
  ПЕРВ=nullptr;
  char ch;
  cin.get(ch);
  while (ch!='\n') {
    ТЕК = new ЭЛМНТ;
    ТЕК->ПРАВ = ПЕРВ;
    ТЕК->ЛЕВ = nullptr;
    ТЕК->ИНФ = ch;
    ПЕРВ = ТЕК;
    if (ТЕК->ПРАВ != nullptr)
      ТЕК->ПРАВ->ЛЕВ = ТЕК;
    cin.get(ch);
  };
}; // FORM
```

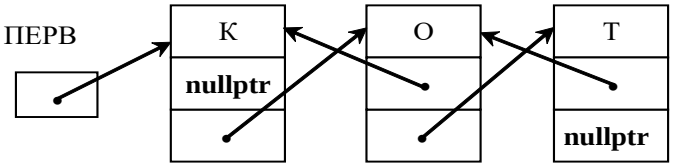

Рис. 10. Результат работы функции FORM при вводе строки «ТОК»

Ниже приведен заголовок функции, осуществляющей удаление элемента, «содержательное» поле которого совпадает с заданным значением. Параметрами функции являются ссылочная переменная – указатель головы списка, и заданный символ.

void УДАЛИТЬ (ЭЛМНТ * &ГОЛОВА, char СИМВОЛ);

3. Индивидуальные задания

Составить программу обработки списка в соответствии с номером задания (табл. 1). Вид списка определяется номером строки, вид обработки – номером столбца. Этот номер столбца определяет вид обработки, указанный в табл. 2. Например, для задания 12 следует рассматривать линейный кольцевой список и произвести (табл. 1, столбец 5) удаление первого элемента, заданного значением информационного поля (табл. 2). В списке должны храниться строки символов, например, фамилии.

Таблица 1

Варианты для реализации

Вид списка	Вид обработки									
	1	2	3	4	5	6	7	8	9	10
Линейный двусвязный	27	3	6	8	11	14	17	21	25	9
Кольцевой односвязный	1	30	7	2	12	15	18	22	19	23
Кольцевой двусвязный	26	28	5	29	10	13	16	20	24	4

В программе предусмотреть подпрограммы формирования списка, вывода, поиска компонента (если требуется), обработки и уничтожения списка. Входные и выходные данные должны передаваться через заголовок. Конец ввода списка задается пустой строкой. Тело главной подпрограммы должно представлять собой последовательность обращений к вспомогательным подпрограммам: формирование списка, контрольный вывод, обработка списка, снова вывод списка, вывод результата, уничтожение списка. Проверку освобождения памяти осуществить подсчетом операторов new и соответствующих им операторов delete, а также контрольным выводом списка.

4. Подготовка к работе

1. Изучить описание работы.
2. Разобраться в приведенных программах.
3. Изобразить структуру данных, указанную в индивидуальном задании, обдумать способ ее обработки.
4. Составить графический алгоритм обработки в виде рисунков, отображающих изменения в структуре списка по аналогии с рисунками в описании работы.
5. Составить программу по этому алгоритму.
6. Разработать тестовые данные и предполагаемые результаты.
7. Подготовить в письменной форме заготовку отчета по работе.

Таблица 2

Способы обработки списков

Номер	Вид обработки
1	Расположить элементы списка в обратном порядке и подсчитать число его элементов. Создаётся новый список.
2	Добавить новый элемент после элемента, заданного ссылочной переменной.
3	Добавить новый элемент после заданного. Элемент задан значением информационного поля. Вставка идёт после последнего найденного элемента с заданным значением информационного поля.
4	Удалить заданный элемент из списка. Удаляемый элемент задан ссылочной переменной.
5	Удалить заданный элемент. Элемент задан значением информационного поля; удаляется первый найденный элемент.
6	Удалить заданный элемент. Элемент задан значением информационного поля; удаляются все такие элементы.
7	Проверить, входит ли заданный элемент в список (результат – булевское значение). Элемент задан значением информационного поля.
8	Подсчитать, сколько имеется элементов с заданным содержимым информационного поля. Вывести порядковые номера найденных элементов.
9	Объединить два списка – второй добавить в хвост первого (кольцевые списки разрываются в произвольном месте)
10	Найти элемент с заданным значением информационного поля. Из функции обработки возвращается значение ссылки, указывающей на найденный элемент, либо nullptr, если заданного элемента в списке нет. Положение элемента для поиска задать дополнительным параметром функции: 0 – первый элемент в списке, 1 - последний.

5. Порядок выполнения работы

1. Ввести тексты подпрограмм формирования и вывода списка. Отладить эти подпрограммы. Проверить с пустым списком, со списком из одного элемента и из трех элементов.

2. Добавить подпрограмму уничтожения списка. Отладить. Проверить с теми же вариантами списков. Вывести список после уничтожения.

3. Добавить обработку списка в соответствии с заданием. Отладить. Проверить на тестовых данных, задающих все варианты обработки. Результаты тестирования записать в отчет.

6. Содержание отчета

1. Подробное индивидуальное задание.

2. Информация в соответствии с подготовкой (пп. 3, 4, 6).

3. Результаты выполнения работы, в том числе и ошибки, и их причины, и способ устранения, и окончательные выходные данные, и время, затраченное на выполнение всей работы.

7. Пример программы

Задание. Составить программу обработки кольцевого односвязного списка с последовательным вызовом подпрограмм: сфомировать список; вывести список; найти первый в списке элемент по значению информационного поля и вернуть его адрес; добавить новый элемент, заданный адресом, в список после найденного элемента; снова вывести список; уничтожить список (задание 30). Для проверки освобождения динамической памяти вывести ее доступный размер в начале и в конце программы, а также вызвать процедуру вывода после уничтожения списка.

```
// program Ezample_List30;
#include <iostream>
#include <cstring>
using namespace std;

typedef
  char Str40[40];   // Имя типа
struct TItem {      // Тип элемента
   Str40 Name;      // Имя
   // Указатель на следующий элемент
   TItem * Next;
};
typedef
  TItem * PItem;     // Указатель на элемент

  PItem List;        // Указатель на список
  Str40 NameFound;   // Имя для поиска элемента
  PItem PNameFound;  // Найденный элемент
  PItem PNewItem;    // Указатель на новый элемент

// Вставить элемент после заданного, на входе - адреса
   void InsertNewItemInCircleList(PItem PredItem, PItem InsItem);
/* Сформировать линейный циклический список,
     вернуть указатель на начало списка */
   void CreateCircleList(PItem &F);
   // Вывести циклический список
   void ShowCircleList(PItem F);
   // Уничтожить список
```

```
void DisposeCircleList(PItem &F);
// Найти адрес элемента с требуемым именем
PItem FoundNameInCircleList(PItem F, Str40 Name);
// Создать вставляемый элемент и вернуть адрес
PItem CreateInsItem();

int main() {
  CreateCircleList(List);
  ShowCircleList(List);

  cout << " Имя для поиска: ";
  cin >> NameFound;
  PNameFound = FoundNameInCircleList(List, NameFound);
  if (PNameFound == nullptr)
    cout << " Имя отсутствует в списке \n";
  else {
    cout << "Имя нашли: " << PNameFound->Name << endl;
    PNewItem = CreateInsItem();
    InsertNewItemInCircleList(PNameFound, PNewItem);
    cout << " Список после вставки нового:\n";
    ShowCircleList(List);
  };
  cout << " Уничтожаем список \n";
  DisposeCircleList(List);
  ShowCircleList(List);   // для проверки
  cin.get();
  return 0;
}

/* Сформировать линейный циклический список,
    вернуть указатель на начало списка */
void CreateCircleList(PItem &F) {
  PItem P;   // Новый элемент
  PItem Last; // Адрес последнего элемента
  Str40 Name;

  F = nullptr; Last = nullptr; // Список пуст
  cout << " Создание циклического списка \n"
          " Вводите имена через Enter,\n"
          " Конец ввода - #\n";
  cin >> Name;          // Читаем Первое имя
  // Создаем линейный список
  // Пока строка не "#"
  while (strcmp(Name, "#")!=0) {
    P = new TItem; // Создать новый элемент
    if (Last == nullptr)
      Last = P;     // Запомнить адрес первого элемента
    strcpy(P->Name, Name); // Сохранить имя
    P->Next = F;   // *P указывает на текущий список

    F = P;          // Новый элемент стал началом списка
    cin >> Name;   // Читаем следующее имя
  };
  if (Last != nullptr)  // Если список не пуст
    Last->Next = F;     // то зациклить его
} // CreateCircleList
```

```
  // Вывести циклический список
void ShowCircleList(PItem F) {
  PItem P;
  cout << " Содержимое списка:\n";
  if (F==nullptr) cout << " Список пуст \n";
  else {
    P=F;  // Берем первый элемент
    do {
      cout << P->Name << endl; // Выводим имя
      P = P->Next;     // Берем следующий элемент
    } while (P!=F);    // Следующий элемент есть первый
  };
} // ShowCircleList

  // Уничтожить список
void DisposeCircleList(PItem &F) {
  PItem P;     // Текущий элемент
  PItem PDel;  // Уничтожаемый элемент
  if (F == nullptr) return; // Список пуст
  P = F;            // Берем первый элемент
  do {
    PDel = P;       // Будем его удалять
    P = P->Next;    // Следующий элемент
    delete PDel;    // Уничтожаем элемент
  } while (P != F); // Всех уничтожили
  F = nullptr;      // Отмечаем - список стал пуст
} // DisposeCircleList

// Найти адрес элемента с требуемым именем
PItem FoundNameInCircleList(PItem F, Str40 Name) {
  bool Found;  // Результат поиска
  PItem P;   // Рассматриваемый элемент
  PItem FoundName = nullptr;
                       // Ничего еще не нашли
  if (F == nullptr) return nullptr; // список пуст
  Found = false;       // Не нашли
  P = F;               // Берем первый элемент
  do {
      // Элемент содержит это имя?
    if (strcmp(P->Name, Name)==0)
      Found = true; // Установить: Нашли элемент
    else
      // Иначе берем следующий элемент
      P = P->Next;
    // Пока не нашли и не просмотрен весь список
  } while (not Found and (P!=F));
  if (Found)                   // Если нашли
    return P; // Вернуть адрес
  return nullptr;
} // FoundNameInCircleList

 // Создать вставляемый элемент и вернуть адрес
PItem CreateInsItem() {
  PItem P; Str40 Name;
  P = nullptr;
  cout << "Вставляемое имя: ";  cin >> Name;
```

```
    P = new TItem;              // Создать элемент
    strcpy(P->Name, Name);     // Заполнить поля
    P->Next = nullptr;
  return P;      // Вернуть адрес
} // CreateInsItem

// Вставить элемент после заданного, на входе - адреса
 void InsertNewItemInCircleList
            (PItem PredItem, PItem InsItem) {
  // Новый указывает на следующий
  InsItem->Next = PredItem->Next;
  // Предыдуший указывает на вставляемый
  PredItem->Next = InsItem;
} // InsertNewItemInCircleList
```