Связные списки

Динамические структуры данных– это структуры данных, память под которые выделяется и освобождается по мере необходимости.

Динамическая структура данных характеризуется тем что:

- ей выделяется память в процессе выполнения программы;
- количество элементов структуры может не фиксироваться;
- размерность структуры может меняться в процессе выполнения программы;

в процессе выполнения программы может меняться характер взаимосвязи между элементами структуры.

Необходимость в динамических структурах данных обычно возникает в следующих случаях.

- Используются переменные, имеющие довольно большой размер (например, массивы большой размерности), необходимые в одних частях программы и совершенно не нужные в других.
- В процессе работы программы нужен массив, список или иная структура, размер которой изменяется в широких пределах и трудно предсказуем.
- Когда размер данных, обрабатываемых в программе, превышает объем сегмента данных.

Достоинства связного представления данных – в возможности обеспечения значительной изменчивости структур:

- размер структуры ограничивается только доступным объемом машинной памяти;
- при изменении логической последовательности элементов структуры требуется не перемещение данных в памяти, а только коррекция указателей;
- большая гибкость структуры.

Вместе с тем, связное представление не лишено и недостатков, основными из которых являются следующие:

- на поля, содержащие указатели для связывания элементов друг с другом, расходуется дополнительная память;
- доступ к элементам связной структуры может быть менее эффективным по времени.

Порядок работы с динамическими структурами данных следующий:

1. создать (отвести место в динамической памяти);
2. работать при помощи указателя;
3. удалить (освободить занятое структурой место).

Классификация динамических структур данных

Во многих задачах требуется использовать данные, у которых конфигурация, размеры и состав могут меняться в процессе выполнения программы. Для их представления используют динамические информационные структуры. К таким структурам относят:

- однонаправленные (односвязные) списки;
- двунаправленные (двусвязные) списки;
- циклические списки;
- бинарные деревья.

Объявление динамических структур данных

Элемент динамической структуры состоит из двух полей:

- информационного поля(поля данных), в котором содержатся те данные, ради которых и создается структура; в общем случае информационное поле само является интегрированной структурой – вектором, массивом, другой динамической структурой и т.п.;
- адресного поля(поля связок), в котором содержатся один или несколько указателей, связывающий данный элемент с другими элементами структуры;
- Объявление элемента динамической структуры данных выглядит следующим образом:

```
struct имя_типа {
                информационное поле;
                адресное поле;
               };
```

Например:

```
struct TNode {
            int Data;//информационное поле
            TNode *Next;//адресное поле
          };
```

Рассмотрим в качестве примера динамическую структуру, схематично указанную на рис.1:

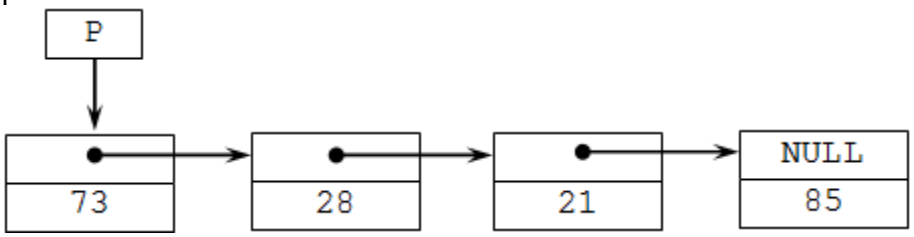


Рис. 1. Схематичное представление динамической структуры

Доступ к данным в динамических структурах

Доступ к динамическим данным выполняется специальным образом с помощью указателей.

Доступ к данным в динамических структурах осуществляется с помощью операции "стрелка" (->), которую называют операцией косвенного выбора элемента структурного объекта, адресуемого указателем. Она обеспечивает доступ к элементу структуры через адресующий ее указатель того же структурного типа. Формат применения данной операции следующий:

УказательНаСтруктуру-> ИмяЭлемента

Операции "стрелка" (->) двуместная. Применяется для доступа к элементу, задаваемому правым операндом, той структуры, которую адресует левый операнд. В качестве левого операнда должен быть указатель на структуру, а в качестве правого – имя элемента этой структуры.

Например:

p->Data;
p->Next;

Работа с памятью при использовании динамических структур

В программах, в которых необходимо использовать динамические структуры данных, работа с памятью происходит стандартным образом. Выделение динамической памяти производится с помощью операции new или с помощью библиотечной функции malloc (calloc). Освобождение динамической памяти осуществляется операцией delete или функцией free.
Например, объявим динамическую структуру данных с именем Node с полями Name, Value и Next, выделим память под указатель на структуру, присвоим значения элементам структуры и освободим память.

```
struct Node {char *Name;
             int Value;
             Node *Next
            };
Node *PNode; //объявляется указатель
PNode = new Node; //выделяется память

PNode->Name = "STO"; //присваиваются значения
PNode->Value = 28;
PNode->Next = NULL;
```

В программах может возникнуть необходимость в использовании динамических структур, если обрабатываются данные, размер которых заранее неизвестен. Память под динамические структуры выделяется в процессе выполнения программы, количество элементов может не фиксироваться и в процессе выполнения программы возможно изменение характера взаимосвязи между элементами структуры. Представление динамических структур в памяти определяется как связное. Каждой динамической структуре ставится в соответствие статическая переменная – ее адрес. Элемент динамической структуры состоит как минимум из двух полей: адресного и информационного.

Списки

Списком называется упорядоченное множество, состоящее из переменного числа элементов, к которым применимы операции включения, исключения. Список, отражающий отношения соседства между элементами, называется *линейным*.

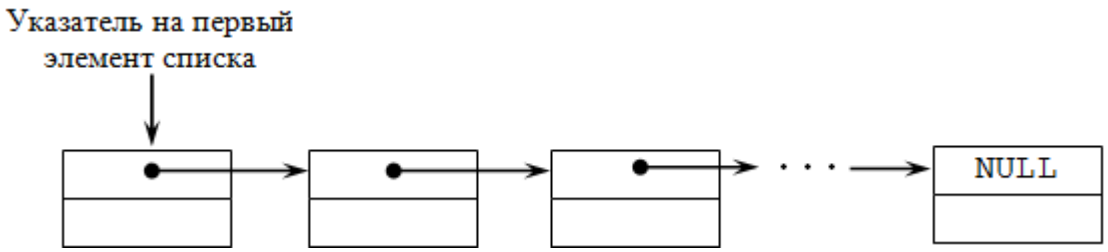
Длина списка равна числу элементов, содержащихся в списке, список нулевой длины называется *пустым списком*. Списки представляют собой способ организации структуры данных, при которой элементы некоторого типа образуют цепочку. Для связывания элементов в списке используют систему указателей.

Каждый список имеет особый элемент, называемый *указателем на начало списка или голову списка*. Голова списка обычно по содержанию отличен от остальных элементов. В поле указателя последнего элемента списка находится специальный признак NULL, свидетельствующий о конце списка.

Линейные связные списки являются простейшими динамическими структурами данных. Из всего многообразия связанных списков можно выделить следующие основные:

- однонаправленные (односвязные) списки;
- двунаправленные (двусвязные) списки;
- циклические (кольцевые) списки.

Однонаправленные (односвязные) списки



Описание простейшего элемента такого списка выглядит следующим образом:

struct имя_типа { информационное поле; адресное поле; };

где информационное поле– это поле любого, ранее объявленного или стандартного, типа;

адресное поле – это указатель на объект того же типа, что и определяемая структура, в него записывается адрес следующего элемента списка.

Например:

```
struct Node {
        int key;//информационное поле
        Node*next;//адресное поле
       };
```

Информационных полей может быть несколько.

Например:

```
struct point {
        char*name;//информационное поле
        int age;//информационное поле
        point*next;//адресное поле
       };
```

Каждый элемент списка содержит ключ, который идентифицирует этот элемент. Ключ обычно бывает либо целым числом, либо строкой.

Основными операциями, осуществляемыми с однонаправленными списками, являются:

- создание списка;
- печать (просмотр) списка;
- вставка элемента в список;
- удаление элемента из списка;
- поиск элемента в списке
- проверка пустоты списка;

- удаление списка.

  Рассмотрим подробнее каждую из приведенных операций.

  Для описания алгоритмов этих основных операций используется следующее объявление:

```
struct SL {//структура данных
                int Data; //информационное поле
                SL *Next; //адресное поле
               };
. . . . . .
SL *Head; //указатель на первый элемент списка
. . . . . .
SL *Current;
//указатель на текущий элемент списка (при необходимости)
```

Создание однонаправленного списка

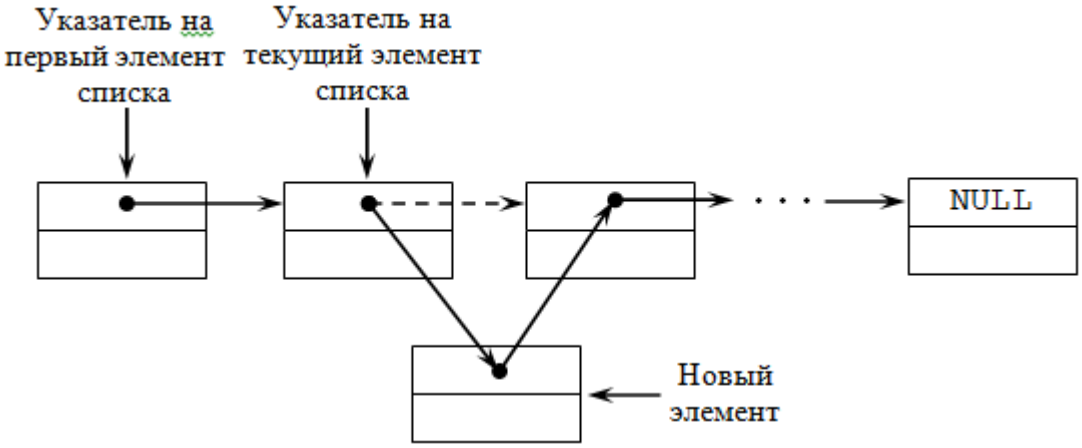
```
//создание однонаправленного списка (добавления в конец)
void Make_Single_List(int n,Single_List** Head){
  if (n > 0) {
    (*Head) = new Single_List();
    //выделяем память под новый элемент
    cout << "Введите значение ";
    cin >> (*Head)->Data;
    //вводим значение информационного поля
    (*Head)->Next=NULL;//обнуление адресного поля
     Make_Single_List(n-1,&((*Head)->Next));
}

}
```

Печать (просмотр) однонаправленного списка

Реализуем данную функцию рекурсивно.

```
//печать однонаправленного списка

void Print_Single_List(Single_List* Head) {
  if (Head != NULL) {
    cout << Head->Data << "\t";
    Print_Single_List(Head->Next);
    //переход к следующему элементу
  }
  else cout << "\n";
}
```

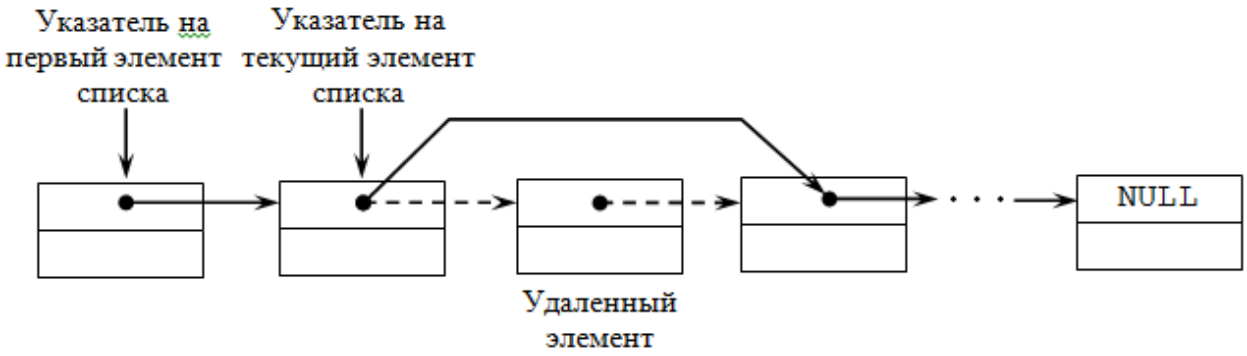
Вставка элемента в однонаправленный список




```
/*вставка элемента с заданным номером в однонаправленный список*/
SL * Insert_Item_ SL (SL * Head,
     int Number, int DataItem){
  Number--;
  SL *NewItem=new(SL);
  NewItem->Data=DataItem;
  NewItem->Next = NULL;
  if (Head == NULL) {//список пуст
    Head = NewItem;//создаем первый элемент списка
  }
  else {//список не пуст
    SL *Current=Head;
    for(int i=1; i < Number && Current->Next!=NULL; i++)
    Current=Current->Next;
    if (Number == 0){
    //вставляем новый элемент на первое место
      NewItem->Next = Head;
      Head = NewItem;
    }
    else {//вставляем новый элемент на непервое место
      if (Current->Next != NULL)
          NewItem->Next = Current->Next;
```

```
      Current->Next = NewItem;
    }
  }
  return Head;
}
```

Удаление элемента из однонаправленного списка




```
/*удаление элемента с заданным номером из однонаправленного списка*/
SL * Delete_Item_Single_List(SL * Head,
      int Number){
  SL *ptr;//вспомогательный указатель
  SL *Current = Head;
  for (int i = 1; i < Number && Current != NULL; i++)
    Current = Current->Next;
  if (Current != NULL){//проверка на корректность
    if (Current == Head){//удаляем первый элемент
      Head = Head->Next;
      delete(Current);
      Current = Head;
    }
    else {//удаляем непервый элемент
      ptr = Head;
      while (ptr->Next != Current)
        ptr = ptr->Next;
      ptr->Next = Current->Next;
```

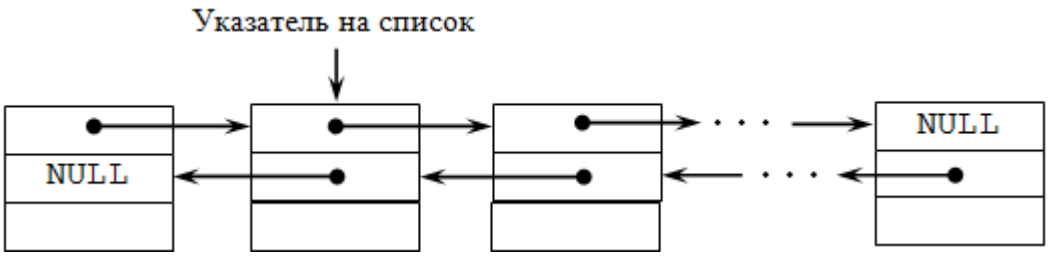
```
      delete(Current);
      Current=ptr;
    }
  }
  return Head;
}
```

Удаление однонаправленного списка

```
/*освобождение памяти, выделенной под однонаправленный список*/
void Delete_ SL (SL * Head){
  if (Head != NULL){
    Delete_ SL (Head->Next);
    delete Head;
  }
}
```

Двунаправленные (двусвязные) списки

Двунаправленный (двусвязный) список– это структура данных, состоящая из последовательности элементов, каждый из которых содержит информационную часть и два указателя на соседние элементы. При этом два соседних элемента должны содержать взаимные ссылки друг на друга.



Описание простейшего элемента такого списка выглядит следующим образом:

```
struct имя_типа {
        информационное поле;
        адресное поле 1;
        адресное поле 2;
```

```
};
```

Где информационное поле – это поле любого, ранее объявленного или стандартного, типа;

адресное поле 1 – это указатель на объект того же типа, что и определяемая структура, в него записывается адресследующего элемента списка ;

адресное поле 2 – это указатель на объект того же типа, что и определяемая структура, в него записывается адреспредыдущего элемента списка.

Например:

```
struct list {
        type elem ;
        list *next, *pred ;
       }
list *headlist ;
```

где type – тип информационного поля элемента списка;

*next, *pred – указатели на следующий и предыдущий элементы этой структуры соответственно.

Переменная-указатель headlist задает список как единый программный объект, ее значение – указатель на первый (или заглавный) элемент списка.

Основные операции, осуществляемые с двунаправленными списками:

- создание списка;
- печать (просмотр) списка;
- вставка элемента в список;
- удаление элемента из списка;
- поиск элемента в списке;
- проверка пустоты списка;
- удаление списка.

Для описания алгоритмов этих основных операций используется следующее объявление:

```
struct DList {//структура данных
        int Data; //информационное поле
        DList *Next, //адресное поле
              *Prior; //адресное поле
       };
. . . . . .
DList *Head; //указатель на первый элемент списка
```

```
    . . . . . .
    DList *Current;
    //указатель на текущий элемент списка (при необходимости)
```

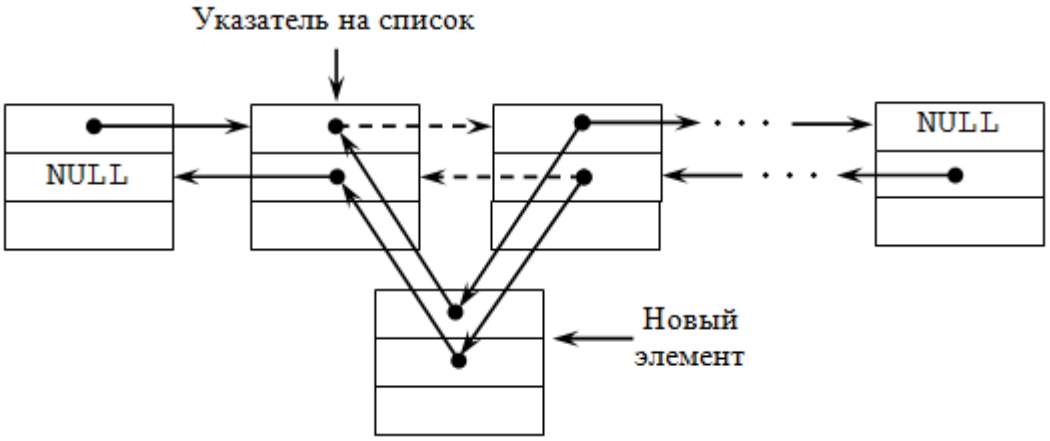
Создание двунаправленного списка

```
//создание двунаправленного списка (добавления в конец)
void Make_ DList (int n, DList ** Head,
        DList * Prior){
  if (n > 0) {
    (*Head) = new DList;
     //выделяем память под новый элемент
    cout << "Введите значение ";
    cin >> (*Head)->Data;
    //вводим значение информационного поля
    (*Head)->Prior = Prior;
    (*Head)->Next=NULL;//обнуление адресного поля
    Make_ DList (n-1,&((*Head)->Next),(*Head));
  }
  else (*Head) = NULL;
}
```

Печать (просмотр) двунаправленного списка

```
//печать двунаправленного списка
void Print_Double_List(DList * Head) {
  if (Head != NULL) {
    cout << Head->Data << "\t";
    Print_ DList (Head->Next);
    //переход к следующему элементу
  }
  else cout << "\n";
}
```

Вставка элемента в двунаправленный список

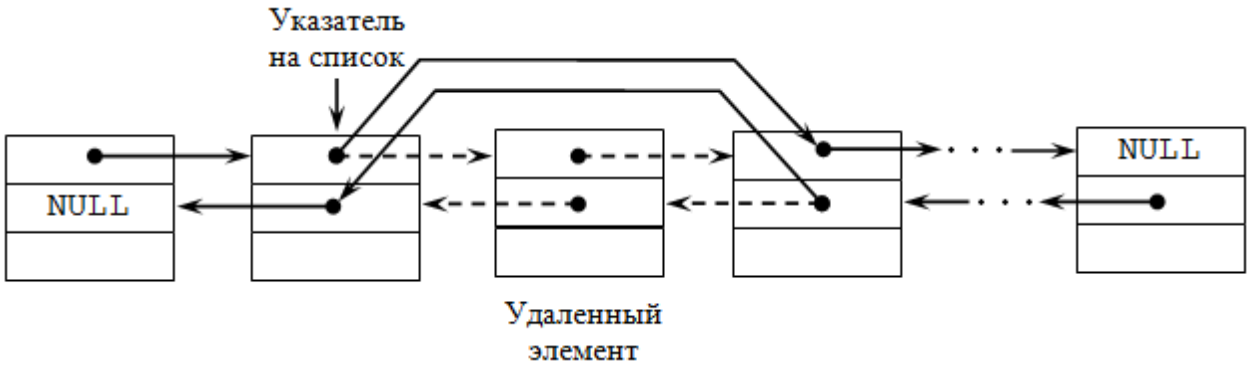


Добавление элемента в двунаправленный список

```
//вставка элемента с заданным номером в двунаправленный список
Double_List* Insert_Item_ DList (DList * Head,
      int Number, int DataItem){
  Number--;
  DList *NewItem=new(DList);
  NewItem->Data=DataItem;
  NewItem->Prior=NULL;
  NewItem->Next = NULL;
  if (Head == NULL) {//список пуст
    Head = NewItem;
  }
  else {//список не пуст
    DList *Current=Head;
    for(int i=1; i < Number && Current->Next!=NULL; i++)
    Current=Current->Next;
    if (Number == 0){
    //вставляем новый элемент на первое место
      NewItem->Next = Head;
```

```
      Head->Prior = NewItem;
      Head = NewItem;
    }
    else {//вставляем новый элемент на непервое место
      if (Current->Next != NULL) Current->Next->Prior = NewItem;
      NewItem->Next = Current->Next;
      Current->Next = NewItem;
           NewItem->Prior = Current;
      Current = NewItem;
    }
  }
  return Head;
}
```

Удаление элемента из двунаправленного списка




```
/*удаление элемента с заданным номером из двунаправленного списка*/
Double_List* Delete_Item_ DList (DList * Head,
      int Number){
  DList *ptr;//вспомогательный указатель
  DList *Current = Head;
  for (int i = 1; i < Number && Current != NULL; i++)
    Current = Current->Next;
  if (Current != NULL){//проверка на корректность
    if (Current->Prior == NULL){//удаляем первый элемент
```

```
      Head = Head->Next;
      delete(Current);
      Head->Prior = NULL;
      Current = Head;
    }
    else {//удаляем непервый элемент
      if (Current->Next == NULL) {
      //удаляем последний элемент
        Current->Prior->Next = NULL;
        delete(Current);
        Current = Head;
      }
      else {//удаляем непервый и непоследний элемент
        ptr = Current->Next;
        Current->Prior->Next =Current->Next;
        Current->Next->Prior =Current->Prior;
        delete(Current);
        Current = ptr;
      }
    }
  }
  return Head;
}
```

Поиск элемента в двунаправленном списке

а) просматривая элементы от начала к концу списка;

б) просматривая элементы от конца списка к началу;

в) просматривая список в обоих направлениях одновременно: от начала к середине списка и от конца к середине (учитывая, что элементов в списке может быть четное или нечетное количество).

Проверка пустоты двунаправленного списка

Операция проверки двунаправленного списка на пустоту осуществляется аналогично проверки однонаправленного списка.

```
//проверка пустоты двунаправленого списка

bool Empty_ DList (DList * Head){

  if (Head!=NULL)  return false;

  else  return true;

}
```

Удаление двунаправленного списка

Операция удаления двунаправленного списка реализуется аналогично удалению однонаправленного списка.

```
//освобождение памяти, выделенной под двунаправленный список

void Delete_ DList (DList * Head){

  if (Head != NULL){

    Delete_ DList (Head->Next);

    delete Head;

  }

}
```

Пример 1. N -натуральных чисел являются элементами двунаправленного списка L, вычислить: X1*Xn+X2*Xn-1+...+Xn*X1. Вывести на экран каждое произведение и итоговую сумму.

Алгоритм:

1. Создаём структуру.
2. Формируем список целых чисел.
3. Продвигаемся по списку: от начала к концу и от конца к началу в одном цикле, перемножаем данные, содержащиеся в соответствующих элементах списка.
4. Суммируем полученные результаты.
5. Выводим на печать

Создание структуры, формирование списка и вывод на печать рассмотрены ранее. Приведем функции для реализации продвижения по списку в обоих направлениях и нахождения итоговой суммы.

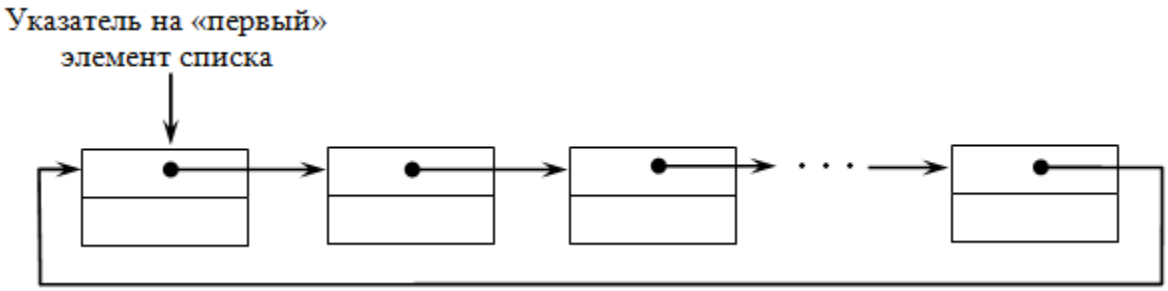
```
//поиск последнего элемента списка

Double_List* Find_End_Item_ DList (DList * Head){

  DList *ptr; //дополнительный указатель
```

```
  ptr = Head;
  while (ptr->Next != NULL){
    ptr = ptr->Next;
  }
  return ptr;
}
```

Поиск итоговой суммы произведений представить программно самостоятельно.

Циклические (кольцевые) списки

Циклический однонаправленный список:



Основные операции, осуществляемые с циклическим однонаправленным списком:

- создание списка;
- печать (просмотр) списка;
- вставка элемента в список;
- удаление элемента из списка;
- поиск элемента в списке;
- проверка пустоты списка;
- удаление списка.

Для описания алгоритмов этих основных операций будем использовать те же объявления, что и для линейного однонаправленного списка.
Приведем в качестве примера функции некоторых из перечисленных основных операций при работе с циклическим однонаправленным списком.
//создание циклического однонаправленного списка

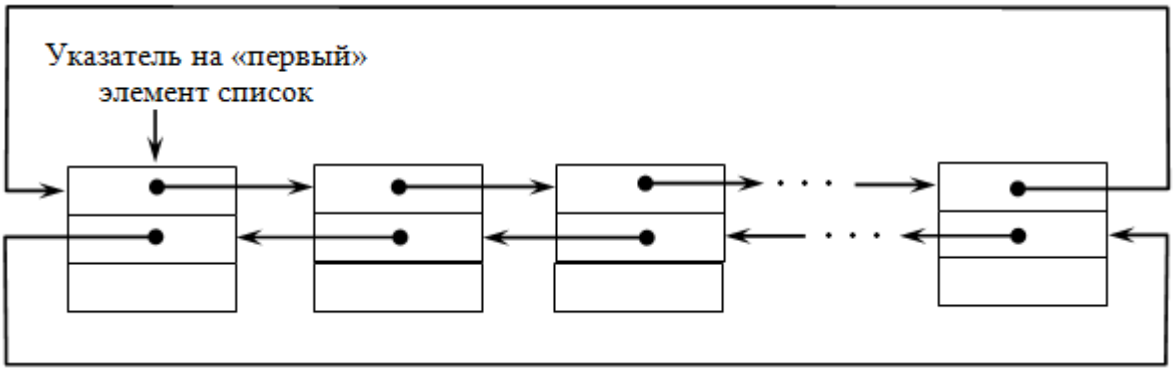
```
void Make_Circle_SL(int n,
      Circle_SL ** Head,Circle_SL * Loop){
   if (n > 0) {
      (*Head) = new Circle_SL ();
      //выделяем память под новый элемент
```

```
        if (Loop == NULL) Loop = (*Head);
        cout << "Введите значение ";
        cin >> (*Head)->Data;
        //вводим значение информационного поля
        (*Head)->Next=NULL;//обнуление адресного поля
        Make_Circle_SL (n-1,&((*Head)->Next),Loop);
    }
    else {
        (*Head) = Loop;
    }
}

/*вставка элемента после заданного номера в циклический однонаправленный список*/
Circle_Single_List* Insert_Item_Circle_SL (Circle_SL * Head,
      int Number, int DataItem){
  Circle_SL *Current = Head;
  //встали на первый элемент
  Circle_SL *NewItem = new(Circle_SL);
  //создали новый элемент
  NewItem->Data = DataItem;
  if (Head == NULL) {//список пуст
    NewItem->Next = NewItem;
    Head = NewItem;
  }
  else {//список не пуст
    for (int i = 1; i < Number; i++)
      Current = Current->Next;
    NewItem->Next = Current->Next;
    Current->Next = NewItem;
  }
  return Head;
```

```
}
//поиск элемента в циклическом однонаправленном списке
bool Find_Item_ Circle_SL (Circle_SL * Head,
        int DataItem){
  Circle_SL *ptr = Head;
  //вспомогательный указатель
  do {
    if (DataItem == ptr->Data) return true;
    else ptr = ptr->Next;
    }
  while (ptr != Head);
  return false;
}

//удаление циклического однонаправленного списка
void Delete_ Circle_SL (Circle_SL * Head){
  if (Head != NULL){
    Head = Delete_Item_ Circle_SL (Head, 1);
    Delete_ Circle_SL (Head);
  }
}
```

Циклический двунаправленный список

Циклический двунаправленный список похож на линейный двунаправленный список, но любой его элемент имеет два указателя, один из которых указывает на следующий элемент в списке, а второй указывает на предыдущий элемент.



Основные операции, осуществляемые с циклическим двунаправленным списком:

- создание списка;
- печать (просмотр) списка;
- вставка элемента в список;
- удаление элемента из списка;
- поиск элемента в списке
- проверка пустоты списка;
- удаление списка.

Для описания алгоритмов этих основных операций будем использовать те же объявления, что и для линейного двунаправленного списка.
Приведем некоторые функции перечисленных основных операций при работе с циклическим двунаправленным списком.
/*вставка элемента после заданного номера в циклический двунаправленный список*/

```
Circle_DL * Insert_Item_Circle_DL

  (Circle_DL * Head, int Number, int DataItem){

 Circle_DL *Current = Head;

 //встали на первый элемент

 Circle_DL *NewItem = new(Circle_DL);

 //создали новый элемент

 NewItem->Data = DataItem;

 if (Head == NULL) {//список пуст

   NewItem->Next = NewItem;

   NewItem->Prior = NewItem;

   Head = NewItem;

 }

 else {//список не пуст

   for (int i = 1; i < Number; i++)

     Current = Current->Next;
```

```
    NewItem->Next = Current->Next;
    Current->Next = NewItem;
    NewItem->Prior = Current;
    NewItem->Next->Prior = NewItem;
  }
  return Head;
}

//поиск элемента в циклическом двунаправленном списке
bool Find_Item_ Circle_DL (Circle_DL * Head,
         int DataItem){
  Circle_DL *ptr = Head;
  //вспомогательный указатель
  do {
    if (DataItem == ptr->Data)
      return true;
    else ptr = ptr->Next;
    }
  while (ptr != Head);
  return false;
}
```