```
int n;
cout<<"\n \n";
cout<<"\n elementi massiva b: ";
cout<<"\n ADRESA:              ZNACHENIE: ";:
cout<<"\n  ------------------------------------\n";
cin>>n;
int *b = new int[n]; //выделение памяти
элементам массива b[n]
for (int i=0; i<n; i++)
{
b[i]=random(17); // создание массива b [п]
cout<< "\n"<<&b[i] << " \t  " <<b[i]; // вывод
адресов ячеек памяти и значений
элементов массива b[n]
}
cout<<"\n \n";
cout<<"\n elementi massiva a: ";
cout<<"\n ADRESA:              ZNACHENIE: ";
cout<<"\n  ------------------------------------\n";
int*  a = new int[n]; //выделение памяти
элементам массива  a[n]
for(int i=0; i<n; i++)   {
a[i]=random(17); // создание массива a [n]
cout<< "\n"<<&a[i] << "  \t  " <<a[i]; // вывод
адресов ячеек памяти и значений элементов массива a[n]
}
delete [] a;
delete [] b;
```

```
int n;
cout<<"\n \n»;
cin>>n;
cout<<"\n elementi massiva b: ";
cout<<"\n ADRESA:     ZNACHENIE: ";
cout<<"\n  ------------------------  \n";
int *b = new int[n]; //выделение памяти
элементам массива b[n]
for (int i=0;i<n; i++)
{
b[i]=rand()%17; // создание массива b [п]
cout<< "\n"<<&b[i] << " \t" <<b[i]; // вывод
адресов ячеек памяти и значений
элементов массива b[n]
}
delete [] b;
cout<<"\n \n";
cout<<"\n elementi massiva a: ";
cout<<"\n ADRESA:     ZNACHENIE: ";
cout<<"\n  ----------------------       \n ";
int*  a = new int[n]; //выделение памяти
элементам массива  a[n]
for(int i=0; i<n; i++)   {
a[i]=rand()%17; // создание массива a [n]
cout<< "\n"<<&a[i] << "\t " <<a[i]; // вывод
адресов ячеек памяти и значений элементов
массива a[n]
}
```

Результат выполнения программы, когда нет освобождения динамической памяти с помощью оператора
delete [] b;

```
5

 elementi massiva b:
 ADRESA:            ZNACHENIE:
  ---------------------------

0x38c0750          10
0x38c0754          15
0x38c0758          9
0x38c075c          0
0x38c0760          15


 elementi massiva a:
 ADRESA:            ZNACHENIE:
  ---------------------------

0x38c0770          3
0x38c0774          3
0x38c0778          14
0x38c077c          11
0x38c0780          2
```

Результат выполнения программы, когда освобождается динамическая память с помощью оператора
delete [] b;

```
5

 elementi massiva b:
 ADRESA:            ZNACHENIE:
  ---------------------------

0xaa7e00           10
0xaa7e04           15
0xaa7e08           9
0xaa7e0c           0
0xaa7e10           15


 elementi massiva a:
 ADRESA:            ZNACHENIE:
  ---------------------------

0xaa7e00           3
0xaa7e04           3
0xaa7e08           14
0xaa7e0c           11
0xaa7e10           2
```

Динамические структуры данных

Например, надо проанализировать текст и определить, какие слова и в каком количество в нем встречаются, причем эти слова нужно расставить по алфавиту.

В таких случаях применяют данные особой структуры, которые представляют собой отдельные элементы, связанные с помощью **ссылок**.

Каждый элемент (**узел**) состоит из двух областей памяти: **поля данных** и **ссылок**. Ссылки – это адреса других узлов этого же типа, с которыми данный элемент логически связан. В языке С++ для организации ссылок используются переменные указатели. При добавлении нового узла в такую структуру выделяется новый блок памяти и (с помощью ссылок) устанавливаются связи этого элемента с уже существующими. Для обозначения конечного элемента в цепи используются **нулевые ссылки** (**NULL**).

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.

Информационных полей может быть несколько.

Динамические структуры, по определению, характеризуются отсутствием физической *смежности* элементов структуры в памяти, непостоянством и непредсказуемостью размера (числа элементов) структуры в процессе ее обработки.

Поскольку элементы динамической структуры располагаются по непредсказуемым адресам памяти, *адрес* элемента такой структуры не может быть вычислен из адреса начального или предыдущего элемента. Для установления связи между элементами динамической структуры используются указатели, через которые устанавливаются явные связи между элементами. Такое *представление* данных в памяти называется *связным*.

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

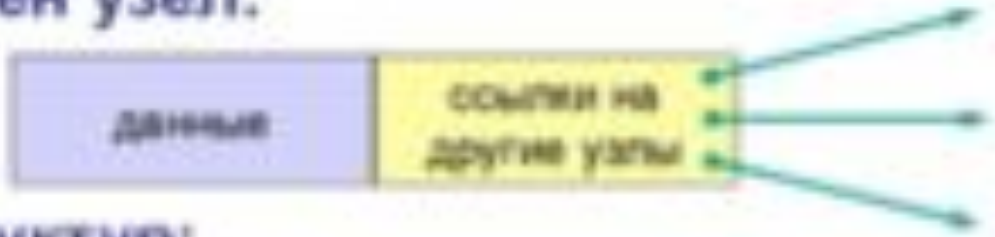
- размер структуры ограничивается только доступным объемом машинной памяти;
- при изменении логической последовательности элементов структуры требуется не перемещение данных в памяти, а только коррекция указателей;
- большая гибкость структуры.

Основные недостатки :

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

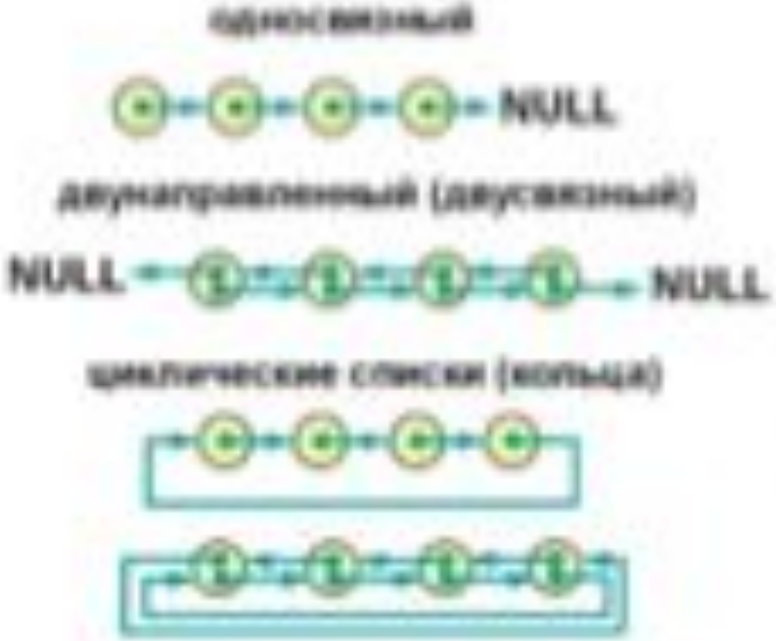
Строение: набор узлов, объединенных с помощью ссылок.

Как устроен узел:

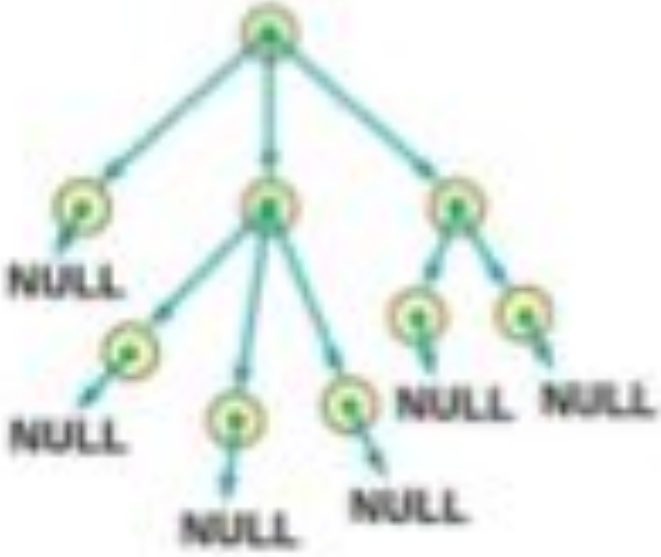


Типы структур:

списки



деревья



графы

Динамическая структура данных характеризуется тем что:

- она не имеет имени;
- 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.

Каждой динамической структуре данных сопоставляется *статическая переменная* типа *указатель* (ее *значение – адрес* этого объекта), посредством которой осуществляется *доступ* к динамической структуре.

Сами динамические величины не требуют описания в программе, поскольку во *время компиляции память* под них не выделяется. Во *время компиляции память* выделяется только под статические величины. Указатели – это статические величины, поэтому они требуют описания.

Для обращения к динамической структуре достаточно хранить в памяти адрес первого элемента структуры. Поскольку каждый элемент динамической структуры хранит адрес следующего за ним элемента, можно, двигаясь от начального элемента по адресам, получить доступ к любому элементу данной структуры.

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

```
struct Single_List {//структура данных
                int Data; //информационное поле
                Single_List *Next; //адресное поле
               };
Single_List *p; //указатель на первый элемент списка
p->Data;
p->Next;
```

Имея возможность явного манипулирования с указателями, которые могут располагаться как вне структуры, так и "внутри" отдельных ее элементов, можно создавать в памяти различные структуры.

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. В минимальном случае, любой элемент линейного списка имеет один указатель, который указывает на следующий элемент в списке или является пустым указателем, что интерпретируется как **конец списка.**
- Структура, элементами которой служат записи с одним и тем же форматом, связанные друг с другом с помощью указателей, хранящихся в самих элементах, называют **связанным списком**. В связанном списке элементы линейно упорядочены, но порядок определяется не номерами, как в массиве, а указателями, входящими в состав элементов списка. Каждый список имеет особый элемент, называемый указателем начала списка (головой списка), который 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.

В основном они отличаются видом взаимосвязи элементов и/или допустимыми операциями.

Однонаправленный (односвязный) список – это структура данных, представляющая собой последовательность элементов, в каждом из которых хранится значение и указатель на следующий элемент списка. В последнем элементе указатель на следующий элемент равен NULL.

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.

Пустой список – это список нулевой длины.

Указатель начала списка (голова списка) – это указатель на первый элемент списка.

Линейный список

В простейшем случае каждый узел содержит всего одну ссылку. Для определенности будем считать, что решается задача частотного анализа текста – определения всех слов, встречающихся в тексте и их количества. В этом случае область данных элемента включает строку (длиной не более 40 символов) и целое число.

Каждый элемент содержит также ссылку на **следующий** за ним элемент. У последнего в списке элемента поле ссылки содержит **NULL**. Чтобы не потерять список, мы должны где-то (в переменной) хранить адрес его первого узла – он называется «головой» списка. В программе надо объявить два новых типа данных – узел списка **Node** и указатель на него **PNode**. Узел представляет собой структуру, которая содержит три поля - строку, целое число и указатель на такой же узел. Правилами языка C++ допускается объявление

```
struct Node {
char word[40]; // область данных
int count;
Node *next; // ссылка на следующий узел
};
typedef Node *PNode; // тип данных: указатель на узел
```

В дальнейшем мы будем считать, что указатель **Head** указывает на начало списка, то есть, объявлен в виде

```
PNode Head = NULL;
```

Первая буква «P» в названии типа **PNode** происходит от слова *pointer* – указатель (англ.) В

начале работы в списке нет ни одного элемента, поэтому в указатель **Head** записывается нулевой адрес **NULL**.

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.

Особое внимание следует обратить на то, что при выполнении любых операций с линейным однонаправленным списком необходимо обеспечивать позиционирование какого-либо указателя на первый элемент. В противном случае часть или весь список будет недоступен.

Создание элемента списка

Для того, чтобы добавить узел к списку, необходимо создать его, то есть выделить память под узел и запомнить адрес выделенного блока. Будем считать, что надо добавить к списку узел, соответствующий новому слову, которое записано в переменной **NewWord.** Составим функцию, которая создает новый узел в памяти и возвращает его адрес. Обратите внимание, что при записи данных в узел используется обращение к полям структуры через указатель.

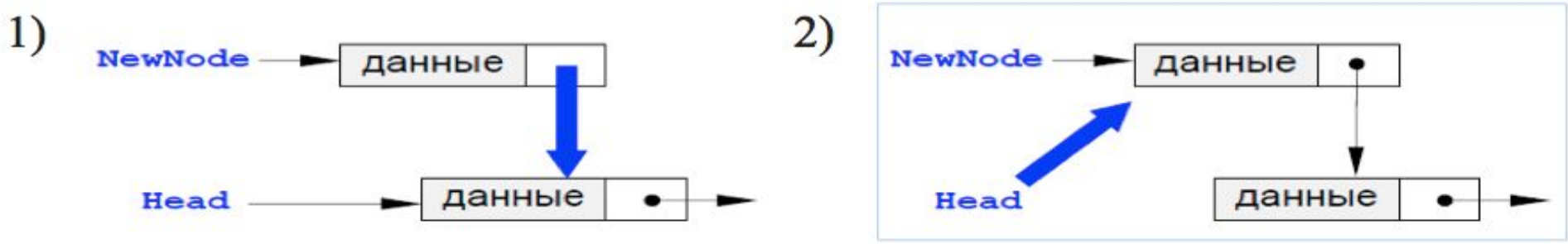
```
PNode CreateNode ( char NewWord[] )
{
PNode NewNode = new Node; // указатель на новый узел
strcpy(NewNode->word, NewWord); // записать слово
NewNode->count = 1; // счетчик слов = 1
NewNode->next = NULL; // следующего узла нет
return NewNode; // результат функции – адрес узла
}
```

После этого узел надо добавить к списку (в начало, в конец или в середину).

Добавление узла

Добавление узла в начало списка

При добавлении нового узла **NewNode** в начало списка надо 1) установить ссылку узла **NewNode** на голову существующего списка и 2) установить голову списка на новый узел



По такой схеме работает процедура **AddFirst**. Предполагается, что адрес начала списка хранится в **Head**. Важно, что здесь и далее адрес начала списка передается *по ссылке*, так как при добавлении нового узла он изменяется внутри функции.

```
void AddFirst (PNode &Head, PNode NewNode)
{
NewNode->next = Head;    // Установить ссылку нового узла на голову списка:
Head = NewNode;  // Установить новый узел как голову списка
}
```

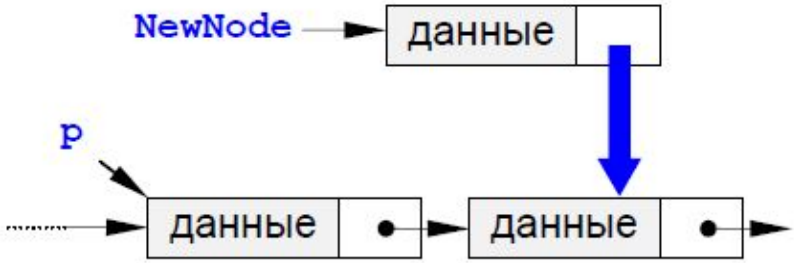
Добавление узла после заданного

Дан адрес **NewNode** нового узла и адрес **p** одного из существующих узлов в списке. Требуется вставить в список новый узел после узла с адресом **p**. Эта операция выполняется в два этапа:

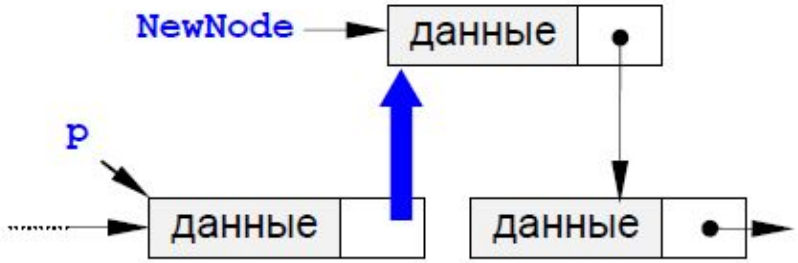
1) установить ссылку нового узла на узел, следующий за данным;

2) установить ссылку данного узла **p** на **NewNode**.

1)



2)



Последовательность операций менять нельзя, потому что если сначала поменять ссылку у узла **p**, будет потерян адрес следующего узла.

```
void AddAfter (PNode p, PNode NewNode)
{
NewNode->next = p->next; // Установить ссылку нового узла на узел, следующий за p
p->next = NewNode; // Установить ссылку узла p на новый узел:
}
```

Добавление узла перед заданным

Эта схема добавления самая сложная. Проблема заключается в том, что в простейшем линейном списке (он называется *односвязным*, потому что связи направлены только в одну сторону) для того, чтобы получить адрес предыдущего узла, нужно пройти весь список сначала.

Задача сведется либо к вставке узла в начало списка (если заданный узел – первый), либо к вставке после заданного узла.

```
void AddBefore(PNode &Head, PNode p, PNode NewNode)
{
PNode q = Head;  // установить вспомогательный указатель q на голову списка;
if (Head == p) {
AddFirst(Head, NewNode); // вставка перед первым узлом
return;
}
while (q && q->next!=p) // ищем узел, за которым следует p
q = q->next; // перейти к следующему узлу
if ( q ) // если нашли такой узел,
AddAfter(q, NewNode); // добавить новый после него
}
```

Такая процедура обеспечивает «защиту от дурака»: если задан узел, не присутствующий в списке, то в конце цикла указатель **q** равен **NULL** и ничего не происходит.

Существует еще один интересный прием: если надо вставить новый узел **NewNode** **до** заданного узла **p**, вставляют узел **после** этого узла, а потом выполняется обмен данными между узлами **NewNode** и **p**. Таким образом, по адресу **p** в самом деле будет расположен узел с новыми данными, а по адресу **NewNode –** с теми данными, которые были в узле **p**, то есть мы решили задачу. Этот прием не сработает, если адрес нового узла **NewNode** запоминается где-то в программе и потом используется, поскольку по этому адресу будут находиться другие данные.

Добавление узла в конец списка

Для решения задачи надо сначала найти последний узел, у которого ссылка равна **NULL**, а затем воспользоваться процедурой вставки после заданного узла. Отдельно надо обработать случай, когда список пуст.

Последовательность операций менять нельзя, потому что если сначала поменять ссылку у узла **p**, будет потерян адрес следующего узла.

```
void AddLast (PNode Head, PNode NewNode)
{
PNode q=Head;
if (Head==NULL)  //если список пуст
{AddFirst(Head,NewNode);  // Добавление узла в начало списка, когда список пуст
return;}
while (q->next) //ищем последний узел
q=q->next;   // перейти к следующему узлу
AddAfter(q,NewNode);// Добавление узла после заданного
 }
```

Проход по списку

Для того, чтобы пройти весь список и сделать что-либо с каждым его элементом, надо начать с

головы и, используя указатель **next**, продвигаться к следующему узлу.

```
PNode p = Head; // начали с головы списка
while ( p != NULL ) { // пока не дошли до конца
// делаем что-нибудь с узлом p
p = p->next; // переходим к следующему узлу
}
```

Поиск узла в списке

Часто требуется найти в списке нужный элемент (его адрес или данные). Надо учесть, что требуемого элемента может и не быть, тогда просмотр заканчивается при достижении конца списка. Такой подход приводит к следующему алгоритму:

1) начать с головы списка;

2) пока текущий элемент существует (указатель – не **NULL**), проверить нужное условие и перейти к следующему элементу;

3) закончить, когда найден требуемый элемент или все элементы списка просмотрены.

Например, следующая функция ищет в списке элемент, соответствующий заданному слову (для которого поле **word** совпадает с заданной строкой **NewWord**), и возвращает его адрес или **NULL**, если такого узла нет.

вход: слово (символьная строка);

выход: адрес узла, содержащего это слово или NULL.

```
PNode Find (PNode Head, char NewWord[])
{
PNode q = Head; // начать с головы списка;
while (q && strcmp(q->word, NewWord))  // пока не дошли до конца списка и слово не равно заданному
q = q->next; // перейти к следующему узлу
return q;
}
```

Вернемся к задаче построения алфавитно-частотного словаря. Для того, чтобы добавить новое слово в нужное место (в алфавитном порядке), требуется найти адрес узла, *перед* которым надо вставить новое слово, так чтобы в списке сохранился алфавитный порядок слов. Это будет первый от начала списка узел, для которого «его» слово окажется «больше», чем новое слово. Поэтому достаточно просто изменить условие в цикле **while** в функции **Find**., учитывая, что функция **strcmp** возвращает «разность» первого и второго слова.

```
PNode FindPlace (PNode Head, char NewWord[])
{
PNode q = Head;
while (q && (strcmp(q->word, NewWord) > 0))
q = q->next; // перейти к следующему узлу
return q;
}
```

Эта функция вернет адрес узла, перед которым надо вставить новое слово (когда функция **strcmp** вернет положительное значение), или **NULL**, если слово надо добавить в конец списка.

Удаление узла

Эта процедура также связана с поиском заданного узла по всему списку, так как нам надо поменять ссылку у предыдущего узла, а перейти к нему непосредственно невозможно. Если мы нашли узел, за которым идет удаляемый узел, надо просто переставить ссылку.

При удалении узла освобождается память, которую он занимал.

Отдельно рассматриваем случай, когда удаляется первый элемент списка. В этом случае адрес удаляемого узла совпадает с адресом головы списка **Head** и надо просто записать в **Head** адрес следующего элемента.

```
void DeleteNode(PNode &Head, PNode OldNode)
{
PNode q = Head;
if (Head == OldNode) // особый случай, когда удаляется первый элемент списка
Head = OldNode->next; // удаляем первый элемент
else {
// ищем предыдущий узел, такой что q->next == p
while (q && q->next != OldNode)
 q = q->next;
if ( q == NULL ) return; // если не нашли, выход
q->next = OldNode->next;
}
delete OldNode; // освобождаем память
}
```

Алфавитно-частотный словарь

Теперь можно полностью написать программу, которая обрабатывает файл **input.txt** и составляет для него алфавитно-частотный словарь в файле **output.txt**.

```
struct Node {
char word[40]; // область данных
int count;
Node *next; // ссылка на следующий узел
};
typedef Node *PNode; // тип данных: указатель на узел
void main()
{
PNode Head = NULL, p, where;
FILE *in, *out;
char word[80];
int n;
in = fopen ( "input. txt", "r" );
while ( 1 ) {
n = fscanf ( in, "%s", word );
 // читаем слово из файла
if ( n <= 0 ) break;
p = Find ( Head, word );
if ( p != NULL ) // если нашли слово,
p->count ++; // увеличить счетчик
else {
p = CreateNode ( word ); // создаем новый узел
where = FindPlace ( Head, word ); // ищем место
if ( ! where )
AddLast ( Head, p );
else AddBefore ( Head, where, p );
      }
}
fclose(in);
out = fopen ( "output. txt", "w" );
p = Head;
while ( p ) { // проход по списку и вывод результатов
fprintf ( out, "%-20s\t%d\n", p->word, p->count );
p = p->next;
}
fclose(out);
}
```

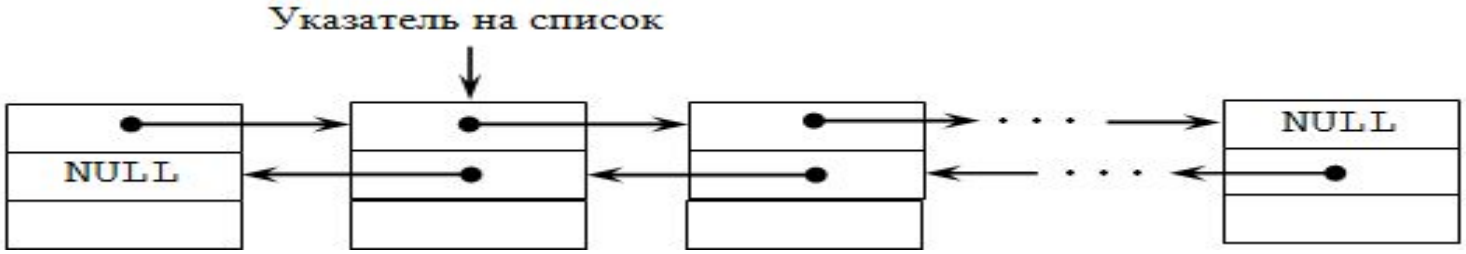
В переменной **n** хранится значение, которое вернула функция **fscanf** (количество удачно прочитанных элементов). Если это число меньше единицы (чтение прошло неудачно или закончились данные в файле), происходит выход из цикла **while**.

Сначала пытаемся искать это слово в списке с помощью функции **Find**. Если нашли – просто увеличиваем счетчик найденного узла. Если слово встретилось впервые, в памяти создается новый узел и заполняется данными. Затем с помощью функции **FindPlace** определяем, перед каким узлом списка надо его добавить.

Когда список готов, открываем файл для вывода и, используя стандартный проход по списку, выводим найденные слова и значения счетчиков.

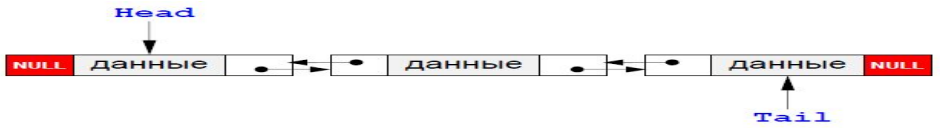
Двусвязный список

Многие проблемы при работе с односвязным списком вызваны тем, что в них невозможно перейти к предыдущему элементу. Возникает естественная идея – хранить в памяти ссылку не только на следующий, но и на предыдущий элемент списка. Для доступа к списку используется не одна переменная-указатель, а две – ссылка на «голову» списк



Каждый узел содержит (кроме полезных данных) также ссылку на **следующий** за ним узел (поле **next**) и предыдущий (поле **prev**). Поле **next** у последнего элемента и поле **prev** у первого содержат **NULL**.

```
struct Node {
  char word[40];  // слово
  int  count;     // счетчик повторений
  Node *next;     // ссылка на следующий элемент
  Node *prev;     // ссылка на предыдущий элемент
  };
typedef Node *PNode; // Указатель на эту структуру:
PNode Head = NULL;  // Адреса «головы» и «хвоста»:
PNode Tail = NULL;
```



Операции с двусвязным списком

Добавление узла в начало списка

При добавлении нового узла **NewNode** в начало списка надо

1) установить ссылку **next** узла **NewNode** на голову существующего списка и его ссылку **prev** в **NULL**;

2) установить ссылку **prev** бывшего первого узла (если он существовал) на **NewNode**;

3) установить голову списка на новый узел;

4) если в списке не было ни одного элемента, хвост списка также устанавливается на новый узел.

По такой схеме работает следующая процедура:

```
void AddFirst(PNode &Head, PNode &Tail, PNode NewNode)
{
NewNode->next = Head;
NewNode->prev = NULL;
if ( Head ) Head->prev = NewNode;
Head = NewNode;
if ( ! Tail ) Tail = Head; // этот элемент – первый
}
```

Добавление узла в конец списка

Благодаря симметрии добавление нового узла **NewNode** в конец списка проходит совершенно аналогично, в процедуре надо везде заменить **Head** на **Tail** и наоборот, а также поменять **prev** и **next**.

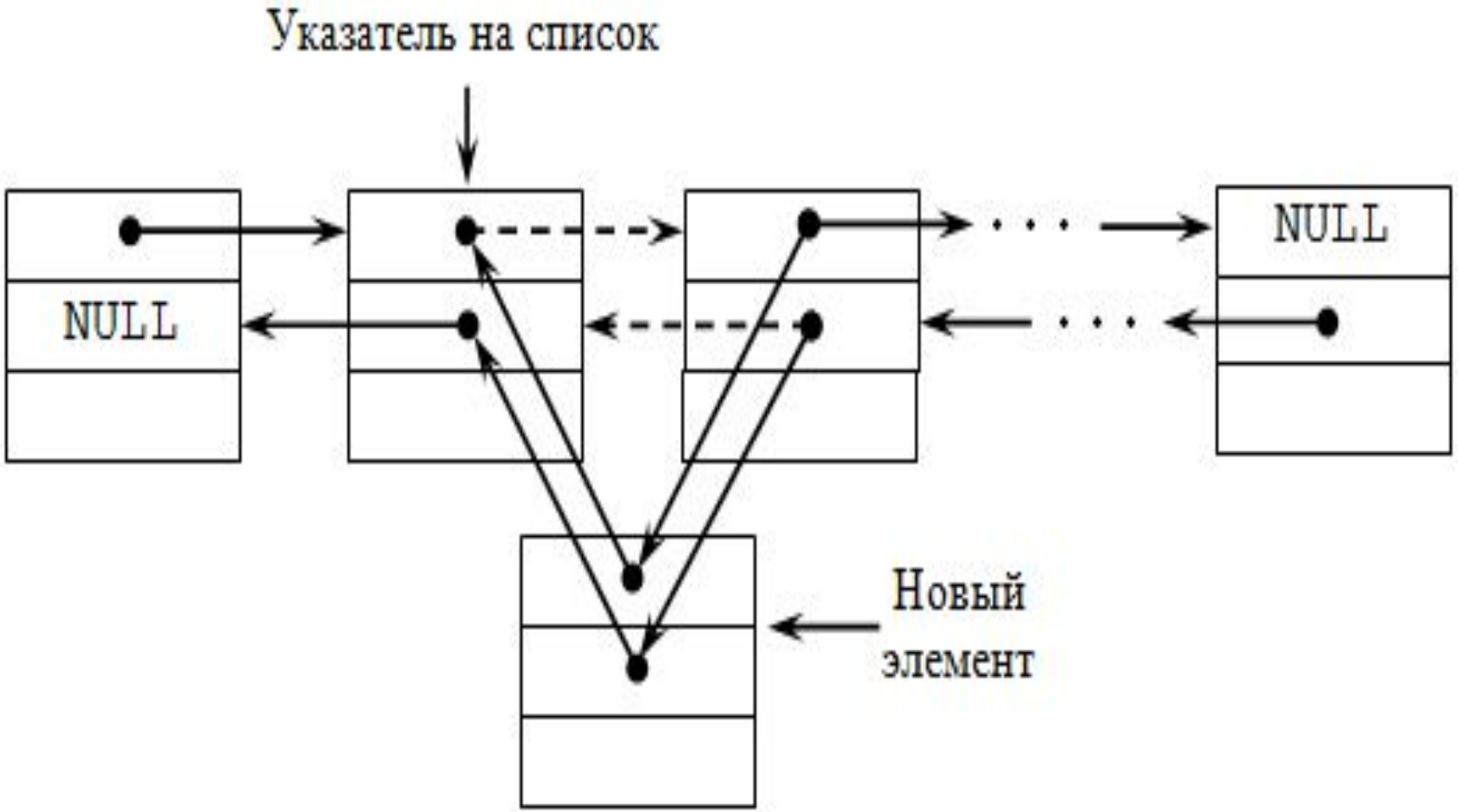
Добавление узла после заданного

Дан адрес **NewNode** нового узла и адрес **p** одного из существующих узлов в списке. Требуется вставить в список новый узел после **p**. Если узел **p** является последним, то операция сводится к добавлению в конец списка (см. выше). Если узел **p** – не последний, то операция вставки выполняется в два этапа:

1) установить ссылки нового узла на следующий за данным (***next***) и предшествующий ему (***prev***);

2) установить ссылки соседних узлов так, чтобы включить **NewNode** в список.

Такой метод реализует приведенная ниже процедура (она учитывает также возможность вставки элемента в конец списка, именно для этого в параметрах передаются ссылки на голову и хвост списка):


Указатель на список
NULL
NULL
Новый
элемент

```
void AddAfter (PNode &Head, PNode &Tail, PNode p,
PNode NewNode)
{
if ( ! p->next )
AddLast (Head, Tail, NewNode); // вставка в конец списка
else {
NewNode->next = p->next; // меняем ссылки нового узла
NewNode->prev = p;
p->next->prev = NewNode; // меняем ссылки соседних узлов
p->next = NewNode;
}
}
```

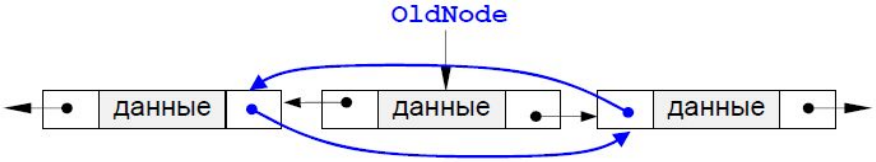
Добавление узла перед заданным выполняется аналогично.

Поиск узла в списке

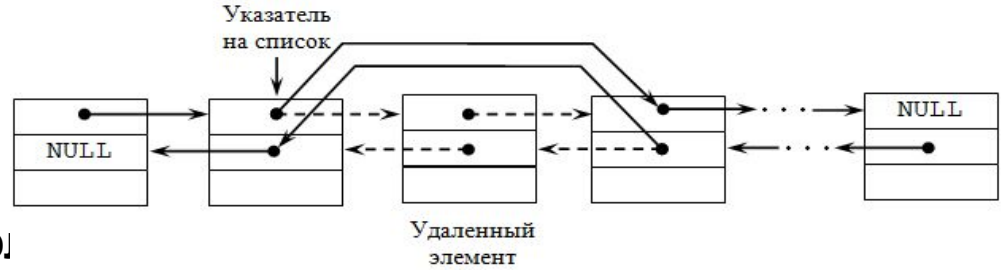
Проход по двусвязному списку может выполняться в двух направлениях – от головы к хвосту (как для

Удаление узла

Эта процедура также требует ссылки на голову и хвост списка, поскольку они могут измениться при удалении крайнего элемента списка. На первом этапе устанавливаются ссылки соседних узлов (если они есть) так, как если бы удаляемого узла не было бы. Затем узел удаляется и память, которую он занимает, освобождается. Эти этапы показаны на рисунке внизу. Отдельно проверяется, не является ли удаляемый узе




```
void Delete(PNode &Head, PNode &Tail, PNode OldNode)
{
if (Head == OldNode) {
Head = OldNode->next; // удаляем первый элемент
if ( Head )
Head->prev = NULL;
else Tail = NULL; // удалили единственный элемент
}
else {
OldNode->prev->next = OldNode->next;
if ( OldNode->next )
OldNode->next->prev = OldNode->prev;
else Tail = NULL; // удалили последний э
}
delete OldNode;
}
```



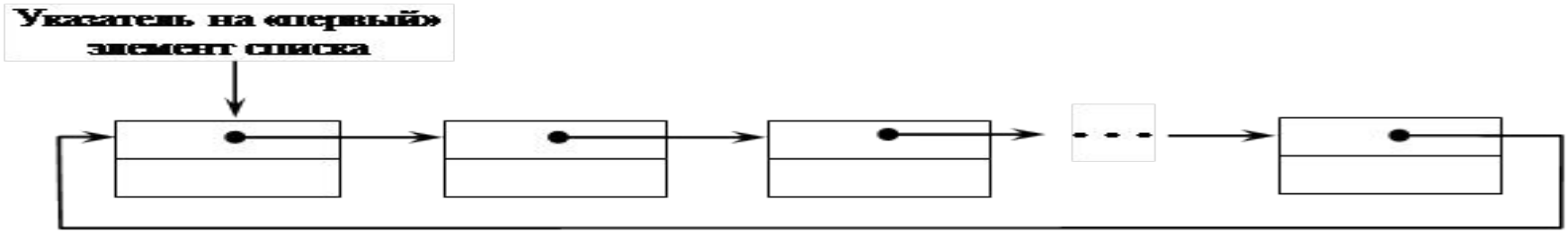
Циклические списки

Иногда список (односвязный или двусвязный) замыкают в кольцо, то есть указатель **next**

последнего элемента указывает на первый элемент, и (для двусвязных списков) указатель **prev** первого элемента указывает на последний. В таких списках понятие «хвоста» списка не имеет

смысла, для работы с ним надо использовать указатель на «голову», причем «головой» можно

считать любой элемент.



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.

В программировании организация работы со списками состоит из двух этапов: *создание и редактирование списка.* Создание списка организуется следующим образом:

- Вначале описывается общая структура элементов списка:

```
struct elem // объявление
{
int d; //поле для хранения данных
 elem* adrK; //объявление поля для хранения
 //адреса следующего элемента списка
elem *adrA; //объявление поля для хранения
 //адреса предыдующего элемента списка
};
```

- во-вторых, добавление нового узла в начало списка

elem * elem_l (int d) // создание первого элемента списка

{

elem *buf= new elem; //выделение памяти под новый элемент списка и передача его адреса в переменную buf

buf->d=d;//передача по данному адресу значения d

buf->adrK=0;//проверка равенства NULL значения следующего или окончания списка

return buf;

}

Добавление нового элемента в конец списка

```
void kosuS(elem** tizSoni,int d)
{
elem *buf=new elem; //выделение памяти для нового
//элемента и передача адресіа в переменную buf
buf->d=d; //передача по данному адресу значения
d
buf->adrA=* tizSoni; //передача новому элементу
адреса предыдущего элемента
(*tizSoni)->adrK=buf; //для предыдущего эелемента
ссылкой на следующий элемент будет адрес
хранящийся в  buf
* tizSoni=buf; //указателю присваиваем адрес
добавленного элемента и переносим в конец
списка
buf->adrK=0; //адрес следующего элемента NULL ии
конец списка
```

-поиск данных в списке:

```
elem* izdeu(elem* const tizBasi, int k)
//берілген элементті тізімнен іздейтін функция
{
elem* buf= tizBasi; //тізімнің басының адресін buf-қа беріп алу
 while(buf) // әзірге buf- тавы адрес NULL болмаса,
{
if (buf->d==k) //сол адрестегі d-ның мәнін берілген k-мен салыстыру
break; //егер, опар сәйкес болса шығып кету
 buf=buf->adrK; //әйтпесе келесі адреске көшіп тексеруді жалғастыру
 }
return buf; //функция қайтаратын мәнге buf-тағы адресті беру
}
```

тізім элементтерінің арасындағы көрсетілген орынға жаңа элементті қосу функциясы, мысалы:

```
elem* ortKosu(elem *const tizBasi, int k, int kl) //жаңа kl элементті тізімнен іздеу
//ізделетін k элементтен кейін орналастыратын функция
{
elem *berElemIzdeu=izdeu(tizBasi,k); //берілген k элементті тізімнен іздеу
if (berElemIzdeu!=0) // k элемент табылса, онда жаңа kl ді
//орналастыру басталады
{
elem* buf = new elem; //жаңа kl -элементке жадыдан орын алып
//адресін buf-қа беру
buf->d=kl;    //жаңа  адреске      kl дің мәнін жіберу
buf->adrK=berElemIzdeu->adrK;  //жаңа элементтің кейінгі
//адресіне табылған элементтің кейінгі адресі беріледі
buf->adrA=berElemIzdeu;      //жаңа  элементтің алдыңғы
//адресіне табылған элементтің өзінің адресі беріледі
 berElemIzdeu->adrK=buf; //табылған элементтің кейінгі адресіне
//жаңа элементтің адресі беріеді.
}
}
```

тізімнен элементті алып тастауды орындайтын функцияны келесі түрде құруға болады, мысалы:

bool aluEl(elem tizBasi, elem** tizSoni,int k)** *//берілген k элементті тізімнен тауып, оны өшіріп тастайтын функцияны* жариялау

{

elem* berElemlzdeu=izdeu(*tizBasi,k); *//* k *үшін орындалатын izdeu функциясының нәтижесін berElemlzdeu -ге беру, егер ол* k *табылса, онда berEl*emlzdeu*-ге сол элементтің адресі, табылмаса ноль беріледі..*

if (berElemlzdeul=0) // k *элемент табылса, онда*

{

if (berElemlzdeu == *tizBasi) *// k элементтің орнын тексеру*

//егер ол тізімнің басында тұрса онда

{

tizBasi = (*tizBasi)->adrK;

(*tizBasi)->adrA=0; *//тізімнің басы* k *-дан кейін тұрған элементке жылжиды*

}

```
else {
if (berElemlzdeu == *tizSoni) //k элементтің орнын тексеру,
//егер ол тізімнің соңында тұрса онда
{
 *tizSoni =(*tizSoni)->adrA;
(*tizSoni)->adrK=0; //тізімнің соңы k-ның алдындат элементке
жылжиды
}
else //егер ол к элемент тізшнің ортасында тұрса онда
{
(berElemlzdeu->adrA)->adrK=berElemlzdeu->adrK;
(berElemlzdeu->adrK)->adrA= berElemlzdeu->adrA;
} //k-ныц алдындағы және соңындағы элементтерді өзара
байланы стыру
delete berElemlzdeu; // k элементі үиіін алынған орынды жадыға
қайтып беру  немесе k элементі тізімнен алып тастау
 return true; //элемент табылып, жойылса функция true мән
қайтарады
}
return false; //элемент табылмаса функция false мән қайтарады
}
```

Программа орындалуының нәтижесі келесі түрде болады:

tizimnin birinshi elementinin mani: 7

tizimnin kelesi 2- gana elementinin mani: -2

tizimnin kelesi 3-gana elementinin mani: 1

tizimnin kelesi 4- gana elementinin mani: 4

tizimnin kelesi 5- gana elementinin mani: 23

TIZIM KURIIDI: 7 -2 1 4 23

izdeitin elementtin manin engizu : -2

izdEl bar =

tabilgan elementten kein koilatin gana elementtin mani: 17

7 -2 17 1 4 23