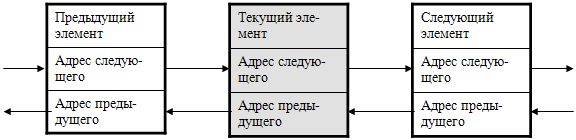
#### Тема 5.Реализация двунаправленных списков

Недостатком рассмотренных выше динамических списковых структур является их однонаправленность – выстраивание логической цепочки элементов только в **одном** направлении - от первого к последнему. Если при обработке списков часто бывает необходимо переходить от текущего элемента к его **предшественнику**, то такая однонаправленная организация становится неудобной: перейти к предшествующему элементу можно только просмотром списка с самого его начала.

В такой ситуации можно перейти к **двунаправленным** спискам, в которых каждый элемент “знает” **обоих** своих соседей, как левого, так и правого. Для этого каждый элемент должен иметь не одно, а **два связующих поля**: указатель на элемент слева и указатель на элемент справа.



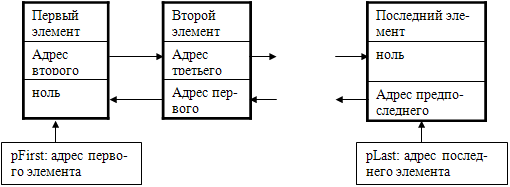
Для двунаправленных списков **расширяется** набор возможных операций за счет добавления поиска и прохода в **обратном** направлении. В частности, поиск в обратном направлении может быть полезен в тех ситуациях, когда **заранее** известно, что искомые элементы находятся где-то в **конце** списка.

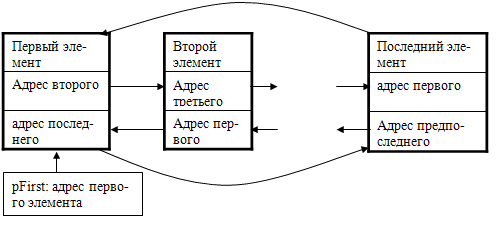
Однако, надо учитывать и некоторые **недостатки** двунаправленных списков, такие как дополнительные (по сравнению с однонаправленными) **затраты памяти** (например – по 4 байта для хранения адреса левого соседа в каждом элементе списка, что для 1 миллиона элементов дает уже около 4 Мб памяти), а также – дополнительные **операции** по изменению указателей при добавлении и удалении элементов.

**Объявления**, которые необходимы для реализации динамического двунаправленного списка, **аналогичны** обычному списку, за исключением наличия у каждого элемента **второго** указательного поля.

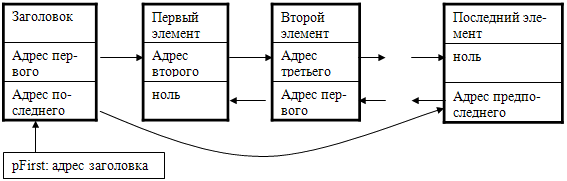
|  |  |
| --- | --- |
| Паскаль | Си |
| **type**  pList2Item = ^TList2Item; TList2Item = **record**                        info : **integer**;                        next : pList2Item;                         prev : pList2Item;                    **end**; **var**    pFirst : pList2Item; | **struct**   List2Item     { **int**     info;       **struct**   List2Item  \*next; **struct**   List2Item  \*prev;     }; **struct**   List2Item   \*pFirst; |

Существует **несколько вариантов** реализации двунаправленных списков:

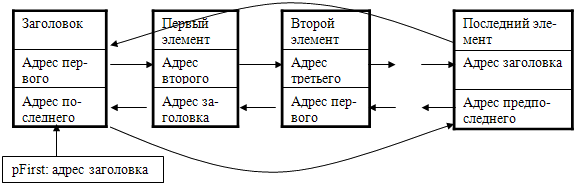
1. Простой список – поле **next** у последнего элемента и поле **prev** у первого содержат пустые адреса, адрес первого элемента задается переменной **pFirst**, адрес последнего – второй основной переменной  **pLast**  
   
2. **Кольцевой (замкнутый)** список: поле **next** у последнего элемента содержит адрес первого элемента, поле **prev** у первого – адрес последнего; здесь достаточно только **одного** основного указателя, например – указателя   **pFirst**  на первый элемент, т.к. из первого элемента сразу можно попасть на последний



1. Список с **заголовочным** элементом – поле **next** заголовка содержит адрес первого элемента, поле **prev** – адрес последнего, для хранения **неизменяемого** адреса заголовка необходима только одна указательная переменная



1. **Кольцевой список с заголовком** – поле **next** у последнего элемента содержит не пустой адрес, а адрес заголовка, также поле **prev** у первого элемента содержит не пустой адрес, а адрес заголовка



Наиболее общей является **последняя** реализация, поэтому далее рассматриваются алгоритмы основных операций именно для **кольцевого** **двунаправленного** списка с **заголовочным** элементом.

**Инициализация пустого списка:**

1. **Выделение** памяти под заголовок с использованием указателя pFirst
2. **Занесение** в адресные поля заголовка **адреса самого заголовка:**

pFirst ^.next := pFirst;   pFirst ^.prev := pFirst;

**Проход по списку в обратном направлении**

1. Объявление вспомогательного указателя  pTemp
2. Установка указателя в адрес **последнего** элемента списка:

pTemp := pFirst^.prev;

1. Организация цикла по условию достижения **первого** элемента

**while**  ( pTemp <> pFirst)  **do**  < телоцикла >

1. В теле цикла обрабатываем текущий элемент и переходим к    предыдущему:

pTemp := pTemp^.prev;

**Внимание! Надо правильно оформлять условие окончания цикла, иначе можно бесконечно проходить по списку с кольцевой структурой!**

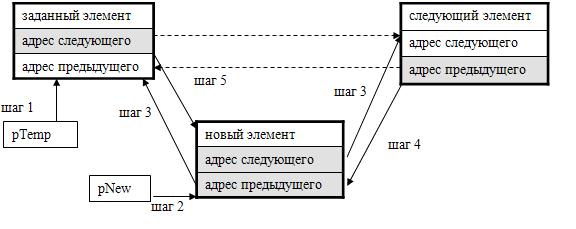
Операции поиска в прямом и обратном направлении реализуются **аналогично**, только с возможностью **досрочного** прерывания цикла при выполнении условия поиска.

**Добавление нового элемента после заданного:**

1. **Поиск** заданного элемента
2. Если элемент найден и адресуется указателем pTemp, то **выделяем** память для нового элемента, используя указатель pNew
3. **Формируем** поля нового элемента, в частности:

* в поле **next** заносится адрес **следующего** элемента (берется из поля **next** найденного элемента)  
  pNew^.next := pTemp^.next;
* в поле **prev** заносится адрес **предшествующего** элемента, которым является найденный элемент  
    pNew^.prev := pTemp;

1. **Изменяем** адресное поле **prev** у элемента, который должен следовать **за** **новым**, на адрес нового элемента  
   pNew^.next^.prev := pNew;  //комментарий дается немного позже
2. **Изменяем** адресное поле **next** у найденного элемента на адрес **нового** элемента  
     pTemp^.next := pNew;



**Комментарий к шагу 4**. На этом шаге надо изменить левый указатель у элемента, **следующего** за заданным, для чего нужен **адрес** этого элемента. Этот адрес можно получить двумя способами:

* либо из поля  **next**  заданного элемента (оно пока еще не изменено) с помощью конструкции  **pTemp^.next**
* либо из установленного на предыдущем шаге поля **next**  у нового элемента (конструкция  **pNew^.next** )

Тогда конструкции  (**pTemp^.next^**)   или  (**pNew^.next^**)  будут обозначать **сам** **адресуемый** элемент-запись, доступ к полям которого можно оформить стандартным образом как    **pTemp^.next^.prev**  или  **pNew^.next^.prev** .  Для упрощения подобных «многоступенчатых» конструкций можно использовать **промежуточный** указатель на следующий элемент, например – с именем  **pNext**.  Это позволит заменить один оператор присваивания (шаг 4) двумя более простыми:

1. pNext  :=  pNew^.next;   //  получение адреса следующего элемента  
   //  а можно и так установить:  pNext  :=  pTemp ^.next;
2. pNext^.prev  :=  pNew;

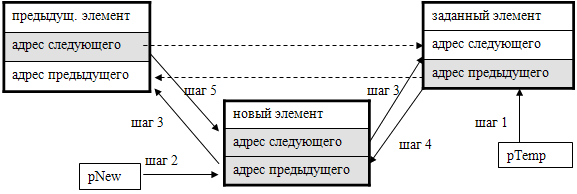
Добавление нового элемента **перед** заданным выполняется аналогично, что объясняется симметричной структурой двунаправленных списков.

**Алгоритм добавления**

1. **Поиск** заданного элемента стандартным образом, **без** определения  его **предшественника** (одно из преимуществ двунаправленного списка)
2. Если элемент найден и адресуется указателем pTemp, то **выделяем** память для нового элемента, используя указатель pNew
3. **Формируем** поля нового элемента, в частности:

* в поле **next** заносится адрес **заданного** элемента  
  pNew^.next := pTemp;
* в поле **prev** заносится адрес **предшествующего** элемента (берется из поля **prev** найденного элемента)  
  pNew^.prev := pTemp^.prev;

1. **Изменяем** адресное поле **prev** у заданного элемента на адрес нового   элемента  
     pTemp^.prev := pNew;
2. **Изменяем** адресное поле **next** у предшествующего элемента на адрес нового элемента  
     pNew^.prev^.next := pNew;    // см. комментарий к предыдущей операции



Наконец, для **удаления** элемента достаточно изменить **правый** указатель у его **левого** соседа и **левый** указатель у его **правого** соседа.

**Удаление заданного элемента**

1. **Проверка** наличия элементов в списке
2. **Поиск** удаляемого элемента (**предшествующий** элемент **не нужен**!)
3. Если элемент найден, **адресуем** его указателем **pTemp**
4. **Изменяем** адресное поле **prev** у следующего за удаляемым элемента на адрес элемента, предшествующего удаляемому (этот адрес берем из поля **prev** удаляемого элемента)  
     pTemp^.next^.prev := pTemp^.prev;    //  опять «многоступенчатость»
5. **Изменяем** адресное поле **next** у элемента, предшествующего удаляемому на адрес элемента, следующего за удаляемым (этот адрес берем из поля **next** удаляемого элемента)  
     pTemp^.prev^.next := pTemp^.next;    //   издесьтоже!
6. **Обрабатываем** удаляемый элемент, используя указатель **pTemp**

