Динамические структуры данных

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

Используется *динамическое распределение памяти*.

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

Порядок работы с *динамическими структурами данных* следующий:

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

Классификация динамических структур данных

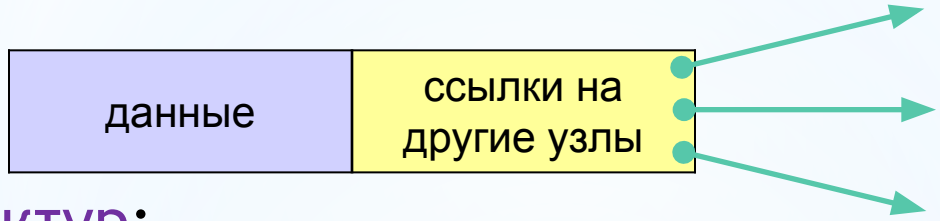
- Списки (*односвязные*, *двусвязные*, *циклические*);
- Стек;
- Дек;
- Очередь;
- Деревья;
- Графы.

Они отличаются способом связи отдельных элементов и/или допустимыми операциями.

Динамические структуры данных

Строение: набор узлов, объединенных с помощью ссылок.

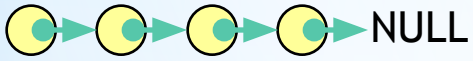
Как устроен узел:



Типы структур:

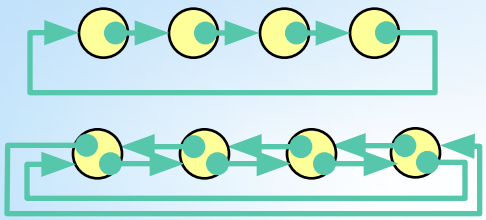
списки

односвязный

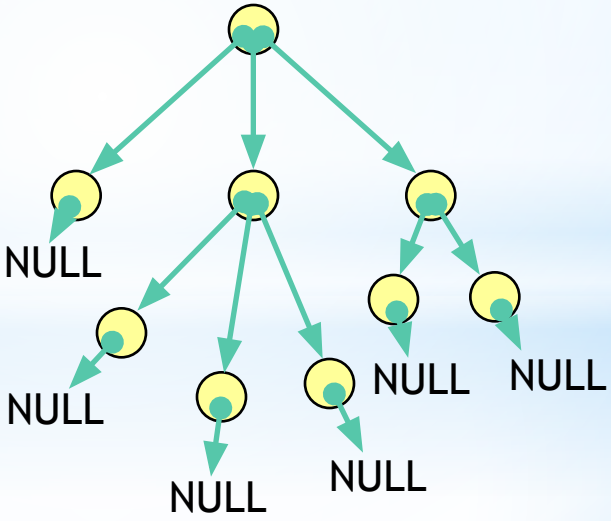


двунаправленный (двусвязный)

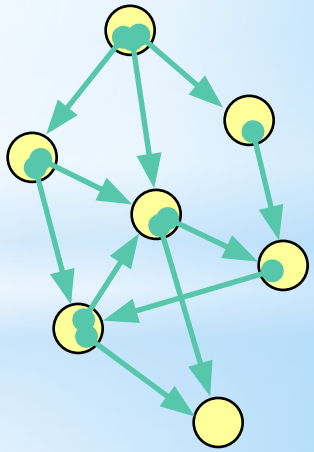
циклические списки (кольца)



деревья



графы



Объявление элемента динамической структуры данных :

struct имя_типа { информационное поле; адресное поле; };

Например:

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

Рекурсивное объявление

Информационных и адресных полей может быть как одно, так и *несколько*.

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.

Формат применения :

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;
```

Указатели на структуры

- Объявление указателя на структуру ничем не отличается от обычного:

```
struct cmplx
{
    double re;
    double im;
};

cmplx c1, c2;
cmplx *pc;
pc = &c1;
```

Указатели на структуры

- Для доступа к членам структуры по указателю на нее можно воспользоваться операцией разыменования:

(*pc).re = 1; cout << c1.re; **// 1**

- Однако, лучше использовать специальную операцию ссылки на член структуры (оператор «стрелка»):

pc->re = 2; cout << c1.re; **// 2**

pc->im = pc->re;

pc=&c2;

pc->re=c1.re;

pc->im=c1.im;

- **Итак, оператор «.» используется для непосредственного обращения к членам структуры, а оператор «->» для доступа к членам структуры через указатель на нее.**

```
struct employee
{
  char name[80];
  int age;
  float wage;
} emp;

struct employee *p = &emp; /* адрес emp заносится в p */
```

для присвоения члену `wage` значения 123.33 необходимо записать

```
emp.wage = 123.23;
```

То же самое можно сделать, использовав указатель на структуру:

```
p->wage = 123.23;
```

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;
delete PNode; // освобождение памяти
```

Программа формирования очереди из 10 элементов и вывода ее на экран

```
#include <iostream>
using namespace std;
void main()
{       struct node {int info;
                struct node *next;
          };
        typedef node *NodePtr;  // указатель на тип node
        NodePtr head = NULL;
        NodePtr p;              // указатель на текущий элемент
        NodePtr tail;           // указатель на "хвост" очереди
        int N = 10;             // количество элементов в очереди
        int cnt = 1;            // счетчик элементов в очереди

        if (head == NULL)
        {       head = new node;
        head->info = cnt++;     // или какому-то другому значению
        head->next = NULL;
        tail = head;
        }
        for (int i = 2; i<=N; i++)
        { p = new node;
        p->info = cnt++;
        tail->next = p;         // в данном случае - NULL
        p->next = NULL;
        tail = p;
        }
// Вывод очереди на экран
p = head;
for (int i = 1; i<=N; i++)
        {       cout << p->info << ' ';
        p = p->next;
        }
cout <<endl;
}
```

Динамические структуры данных:

однонаправленные и двунаправленные списки

Понятие списка хорошо известно из жизненных примеров:

- *список* студентов учебной группы,
- *список* призёров олимпиады,
- *список* (перечень) документов для представления в приёмную комиссию,
- *список* почтовой рассылки,
- *список* литературы для самостоятельного чтения и т.п.

Список – последовательность элементов, связанных посредством ***указателей*** (ссылок)

Элементы списка также называются *узлами*

Размер списка – количество находящихся в нем элементов

Пустой список – список размера 0

Каждый элемент (узел) списка состоит из *двух частей*:

- информационная – содержит значение элемента
- адресная – содержит указатель на тот узел, который связан с данным узлом

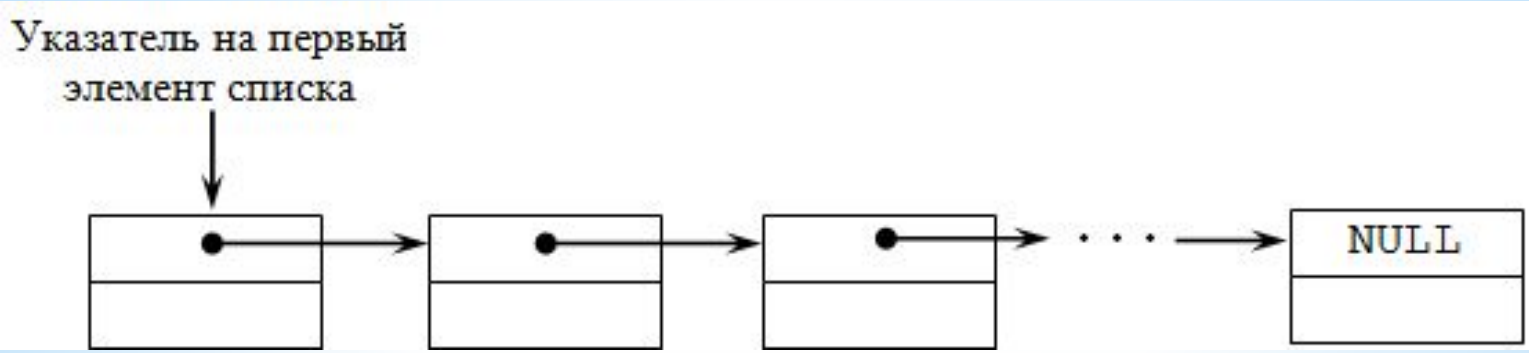
Отсутствие указателя (пустой указатель или значение 0 на месте указателя) означает, что данный элемент является ***последним в списке****.*

Каждый список имеет особый элемент - ***начало списка*** (голова списка), который обычно по содержанию отличен от остальных элементов.

В поле указателя ***последнего элемента списка*** находится специальный признак NULL, свидетельствующий о ***конце списка***.

Наиболее простой динамической структурой является однонаправленный *список*, элементами которого служат объекты *структурного типа*.

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.



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.

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

Для описания алгоритмов этих основных операций используется следующее *объявление*:

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

Создание однонаправленного списка

Для того, чтобы создать *список:*

- создать сначала *первый элемент списка*,
- при помощи функции добавить к нему остальные элементы.

При относительно небольших размерах списка можно использовать рекурсивную функцию. Добавление может выполняться как в начало, так и в конец 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

Операция печати списка заключается в *последовательном* просмотре всех элементов списка и выводе их значений на экран.

Для обработки списка организуется **функция**, в которой нужно переставлять *указатель* на следующий *элемент списка* до тех пор, пока *указатель* не станет равен NULL, то есть будет достигнут конец списка.

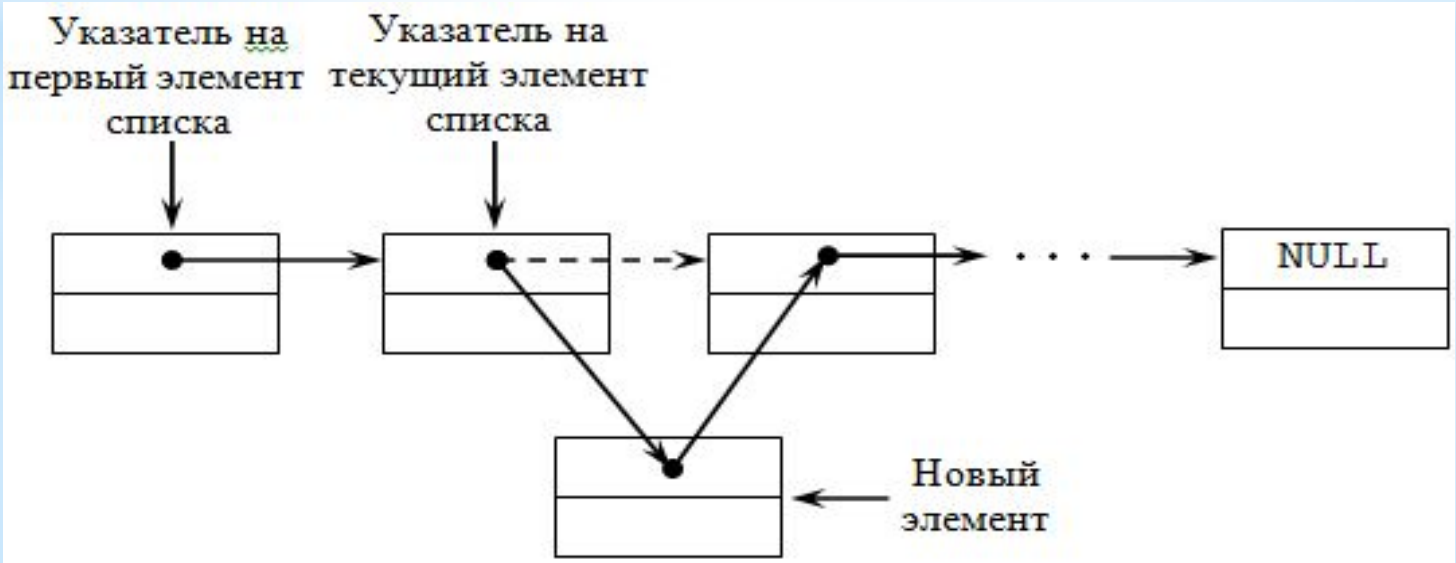
Реализуем данную функцию рекурсивно.

```
//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

В динамические структуры легко добавлять элементы, так как для этого достаточно *изменить значения адресных полей*.

Вставка первого и последующих элементов списка *отличаются* друг от друга. Поэтому в функции, реализующей данную операцию, сначала осуществляется проверка, *на какое место* вставляется элемент. Далее реализуется соответствующий *алгоритм* добавления



```
/*вставка элемента с заданным номером в однонаправленный список*/
Single_List* Insert_Item_Single_List(Single_List* Head,
      int Number, int DataItem){
  Number--;
  Single_List *NewItem=new(Single_List);
  NewItem->Data=DataItem;
  NewItem->Next = NULL;
  if (Head == NULL) {//список пуст
    Head = NewItem;//создаем первый элемент списка
  }
  else {//список не пуст
    Single_List *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;
}
```

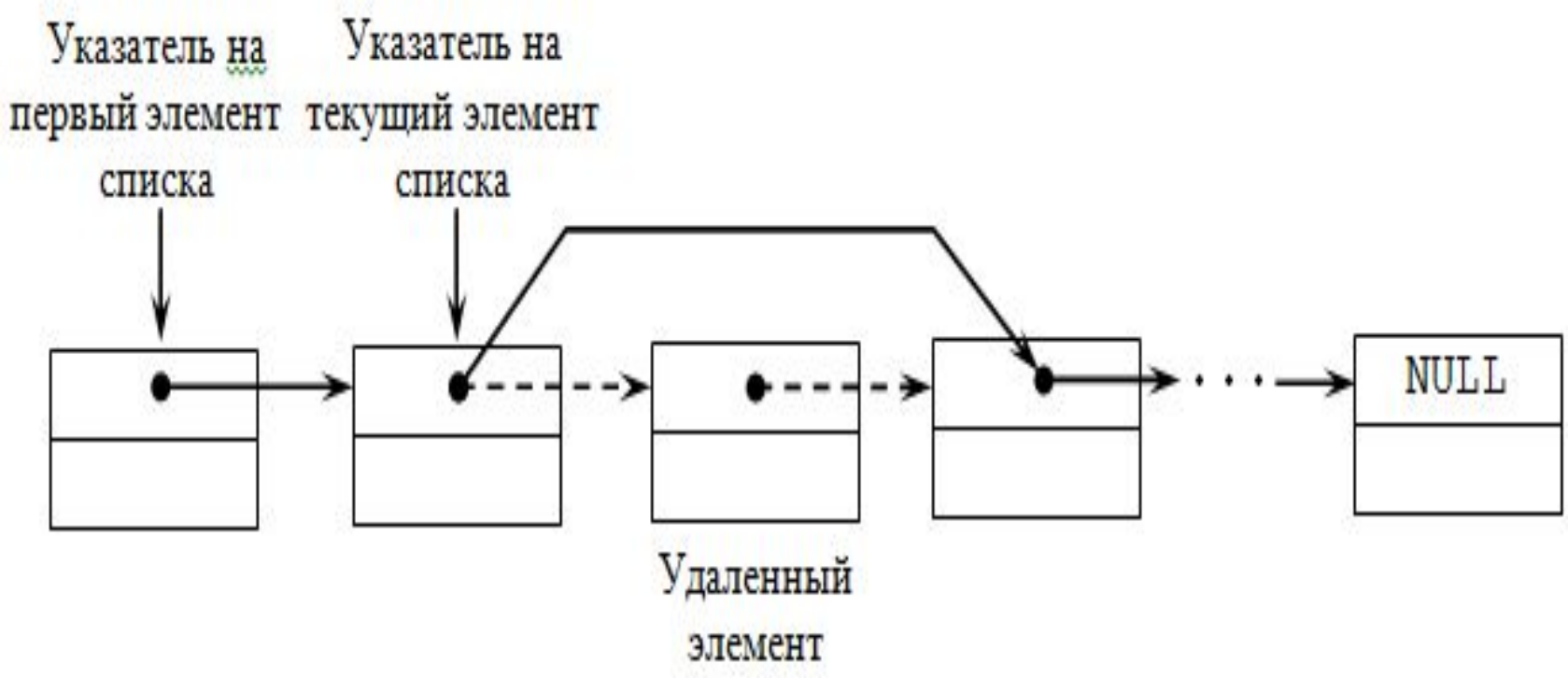
выделить память под структуру, записать ее адрес в переменную NewItem

Удаление элемента из однонаправленного списка

- Из динамических структур можно удалять элементы, так как для этого достаточно *изменить значения адресных полей*.
- Операция удаления элемента однонаправленного списка осуществляет удаление элемента, на который *установлен указатель текущего элемента*.
- После удаления указатель текущего элемента устанавливается на предшествующий элемент списка или на новое начало списка, если удаляется первый.

Алгоритмы удаления первого и последующих элементов списка отличаются друг от друга. Поэтому в функции, реализующей данную операцию, осуществляется проверка, какой элемент удаляется. Далее реализуется соответствующий алгоритм удаления

Удаление элемента из однонаправленного списка



```
/*удаление элемента с заданным номером из однонаправленного списка*/
Single_List* Delete_Item_Single_List(Single_List* Head,
      int Number){
  Single_List *ptr;//вспомогательный указатель
  Single_List *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;
}
```

Поиск элемента в однонаправленном списке

Операция поиска элемента в списке заключается в *последовательном просмотре* всех элементов списка до тех пор, пока текущий элемент не будет содержать *заданное значение* или пока не будет *достигнут конец списка*.

В последнем случае фиксируется *отсутствие* искомого элемента в списке (функция принимает значение false)

```
//поиск элемента в однонаправленном списке
bool Find_Item_Single_List(Single_List* Head, int DataItem){
  Single_List *ptr; //вспомогательным указатель
  ptr = Head;
  while (ptr != NULL){//пока не конец списка
    if (DataItem == ptr->Data) return true;
    else ptr = ptr->Next;
  }
  return false;
}
```

Удаление однонаправленного списка

Операция удаления списка заключается в *освобождении динамической памяти*.

Для данной операции организуется *функция*, в которой нужно переставлять указатель на следующий элемент списка до тех пор, пока указатель не станет равен NULL, то есть не будет достигнут конец списка.

Реализуем рекурсивную функцию.

```
/*освобождение памяти, выделенной под однонаправленный список*/

void Delete_Single_List(Single_List* Head){
  if (Head != NULL){
    Delete_Single_List(Head->Next);
    delete Head;
  }
}
```

Таким образом,

однонаправленный список имеет только **один** указатель в каждом элементе.

Это позволяет **минимизировать расход памяти** на организацию такого списка.

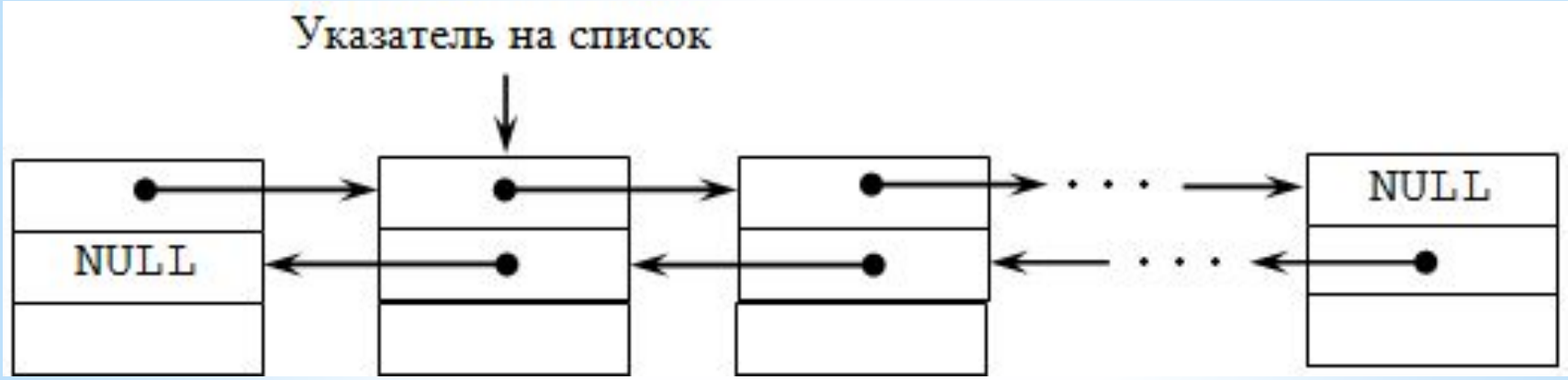
Одновременно это позволяет осуществлять переходы между элементами *только в одном направлении*, что зачастую **увеличивает время**, затрачиваемое на обработку списка.

Например, для перехода к предыдущему элементу необходимо осуществить просмотр списка с начала до элемента, указатель которого установлен на текущий элемент.

Двунаправленные (двусвязные) списки

Для ускорения многих операций целесообразно применять переходы между элементами списка в *обоих направлениях*. Это реализуется с помощью *двунаправленных списков*, которые являются сложной динамической структурой.

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.



Двунаправленные (двусвязные) списки

В таком списке каждый элемент (кроме первого и последнего) связан с предыдущим и следующим за ним элементами.

Каждый элемент двунаправленного списка имеет три поля :

- одно поле содержит *ссылку на следующий элемент*,
- другое поле – *ссылку на предыдущий элемент*
- третье поле – *информационное*.

Наличие ссылок на следующее звено и на предыдущее позволяет двигаться по списку от каждого звена *в любом направлении*: от звена к концу списка или от звена к началу списка, поэтому такой список называют двунаправленным.

Описание простейшего элемента такого списка:

```
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 адрес следующего элемента списка ;
- адресное поле 2 – это указатель на объект того же типа, что и определяемая структура, в него записывается адрес предыдущего 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, выполняемые над двунаправленным списком, те же, что и для однонаправленного списка.

Так как двунаправленный список более гибкий, чем однонаправленный, то при *включении элемента в список*, нужно использовать указатель как на элемент, за которым происходит включение, так и указатель на элемент, перед которым происходит включение.

При исключении элемента из списка нужно использовать как указатель на сам исключаемый элемент, так и указатели на предшествующий или следующий за исключаемым элементы.

Но так как элемент двунаправленного списка имеет два указателя, то при выполнении операций включения/исключения элемента надо изменять больше связей, чем в однонаправленном списке.

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.

Особое внимание следует обратить на то, что в отличие от однонаправленного списка здесь *нет необходимости* обеспечивать позиционирование какого-либо указателя именно *на первый элемент* списка, так как благодаря двум указателям в элементах можно получить доступ к любому элементу списка из любого другого элемента, осуществляя переходы в прямом или обратном направлении.

Однако по правилам хорошего тона программирования указатель желательно ставить на заголовок списка.

Создание двунаправленного списка

Для того, чтобы создать список, нужно создать сначала *первый элемент* списка, а затем при помощи *функции* добавить к нему остальные элементы. Добавление может выполняться как в начало, так и в конец списка. Реализуем рекурсивную функцию.

```
//создание двунаправленного списка (добавления в конец)
void Make_Double_List(int n,Double_List** Head,
        Double_List* Prior){
  if (n > 0) {
    (*Head) = new Double_List();
     //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_Double_List(n-1,&((*Head)->Next),(*Head));
  }
  else (*Head) = NULL;
}
```