

Лекция по курсу  
«Алгоритмы и структуры данных» /  
«Технологии и методы программирования»

Списки

Мясников Е.В.

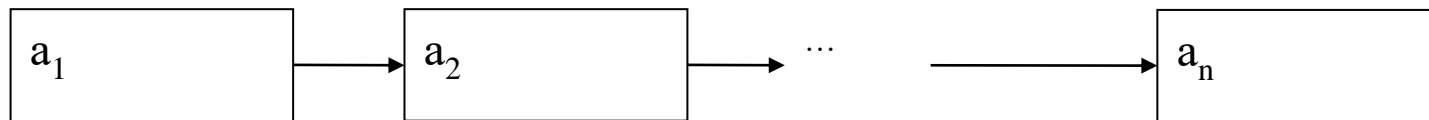
# Список

**Линейным списком** называют последовательность однотипных элементов, возможно, с повторами. Обычно линейный список записывают в следующем виде:

$$L=(a_1, a_2, \dots, a_n),$$

где  $a_i, i=1..n$  – элементы списка.

Графически список обозначают следующим образом:



**Длиной** списка называют количество элементов списка.

При  $n=0$  список пуст.

Первый элемент списка  $a_1$  называют также **головным** элементом списка.

Последний элемент  $a_n$  – **концевым** элементом списка или **хвостом**.

Элементы также называют узлами, объектами.

На множестве элементов  $\{a_i\}, i=1,...,n$ , определены отношения предшествования и следования. Для элемента  $a_i, i=2,...,(n-1)$  существует единственный предшественник  $a_{i-1}$  и единственный последователь  $a_{i+1}$ . Элемент  $a_1$  не имеет предшественника,  $a_n$  не имеет последователя.

# Типовые операции. Способы хранения

При работе со списками используются следующие **типовые операции**:

- обращение к элементу списка,
- поиск элемента списка,
- вставка элемента списка,
- удаление элемента списка,
- упорядочивание элементов списка.

При этом эффективность выполнения приведенных выше операций зависит от способа хранения списка в памяти ЭВМ.

Выделяют два основных способа хранения списков: последовательное хранение и связное хранение списков.

При **последовательном хранении** элементы списка хранятся в памяти последовательно, в смежных областях. Реализуется такой способ с использованием массива.

При **связном хранении** элементы списка представляют собой отдельные объекты (структуры), размещаемые в произвольных областях памяти и связываемые в единую последовательность с использованием специальных полей связи.

# Односвязные списки

При использовании **односвязных списков** для хранения элементов объявляют структуру, имеющую единственное поле связи:

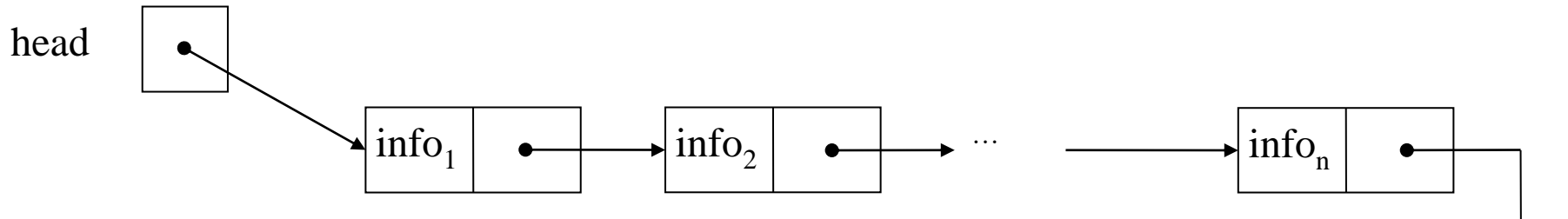
```
struct ListElm{  
    Тип info;  
    ListElm *link;  
};
```

Здесь info – поле некоторого типа Тип, содержащее информацию, хранящуюся в элементе списка,  
link – поле связи со следующим элементом, представляющее собой типизированный указатель.

При связном способе хранения элементов для получения доступа к списку используют указатель на головной элемент:

```
ListElm *head;
```

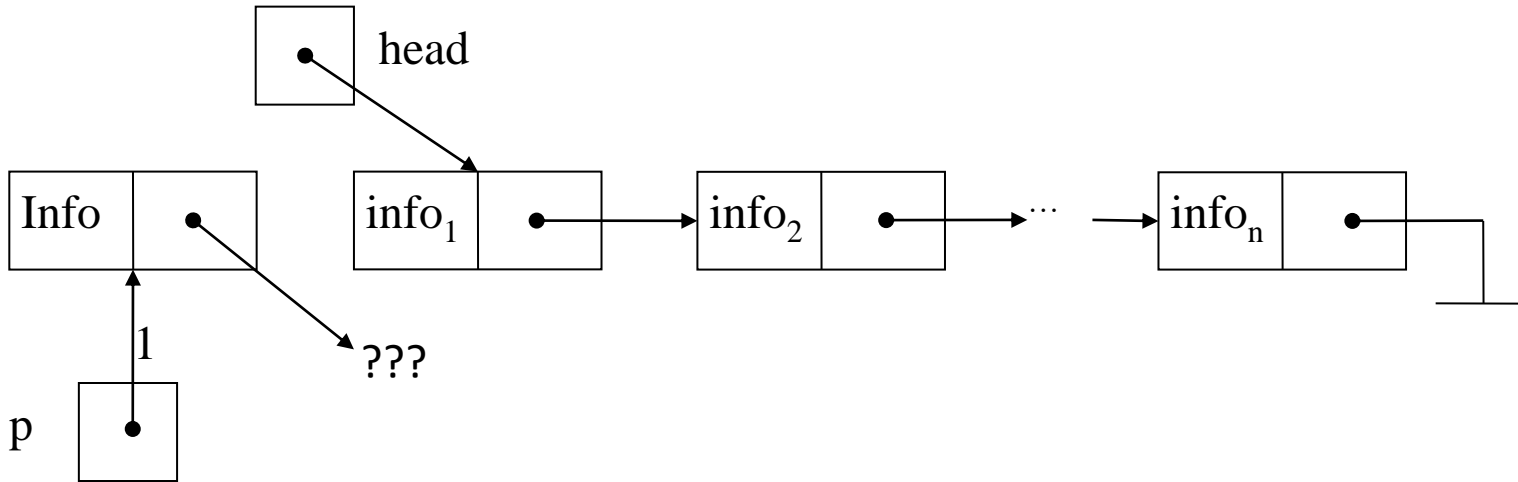
Графически изобразить односвязный список можно следующим образом:



# Односвязные списки.

## Добавление элемента в голову списка

ШАГ 1

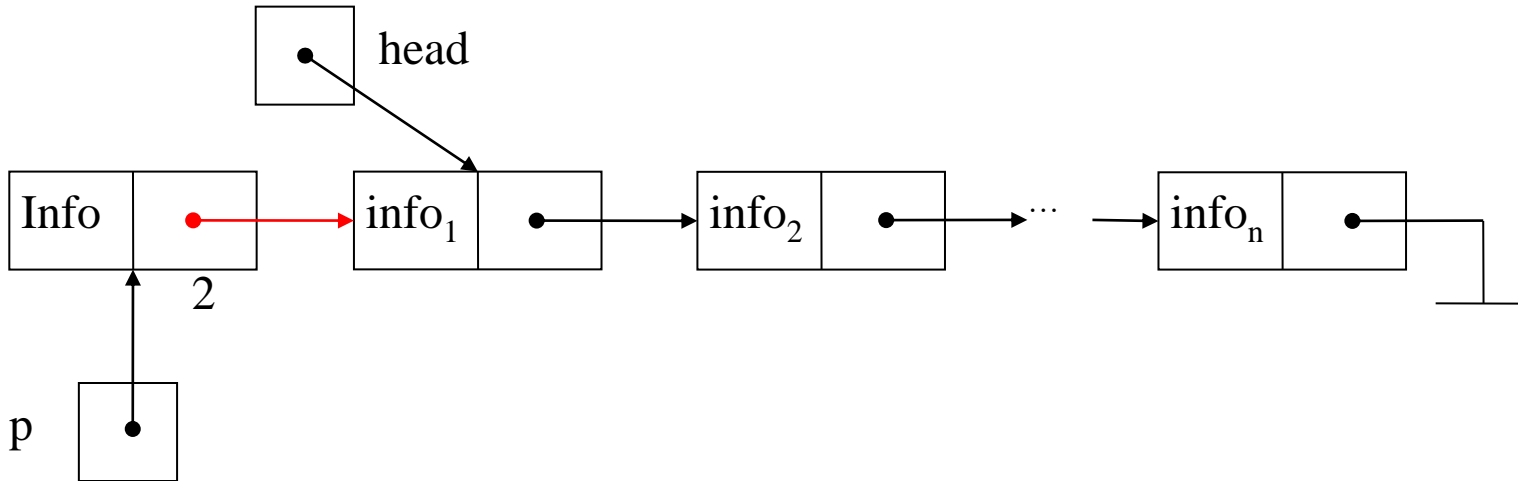


```
ListElm* p =  
    new ListElm; //1  
p->info = Info;  
...
```

# Односвязные списки.

## Добавление элемента в голову списка

ШАГ 2

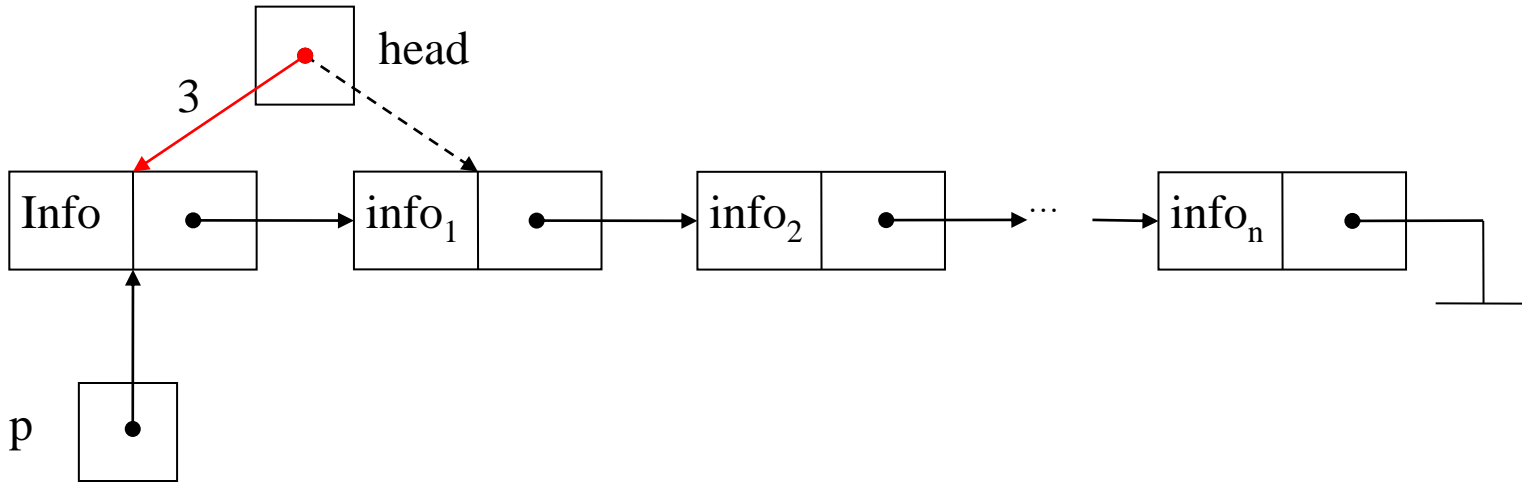


```
ListElm* p =  
    new ListElm; //1  
p->info = Info;  
p->link = head; //2  
...
```

# Односвязные списки.

## Добавление элемента в голову списка

ШАГ 3

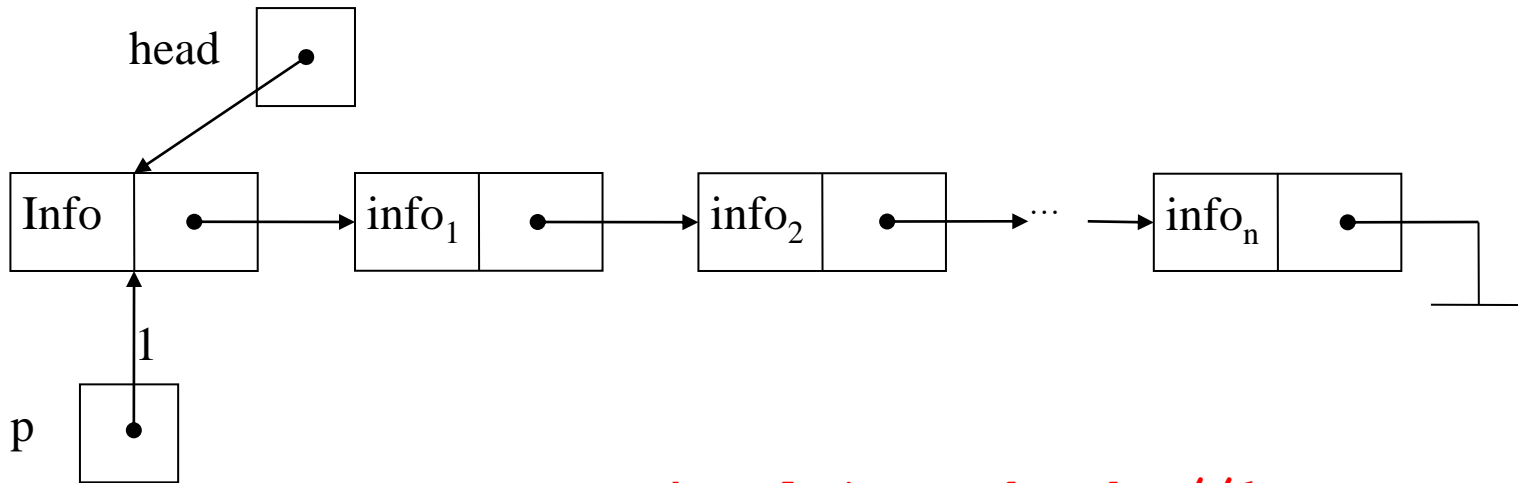


```
ListElm* p =  
    new ListElm; //1  
p->info = Info;  
p->link = head; //2  
head = p;      //3
```

# Односвязные списки.

## Удаление элемента из головы списка

ШАГ 1



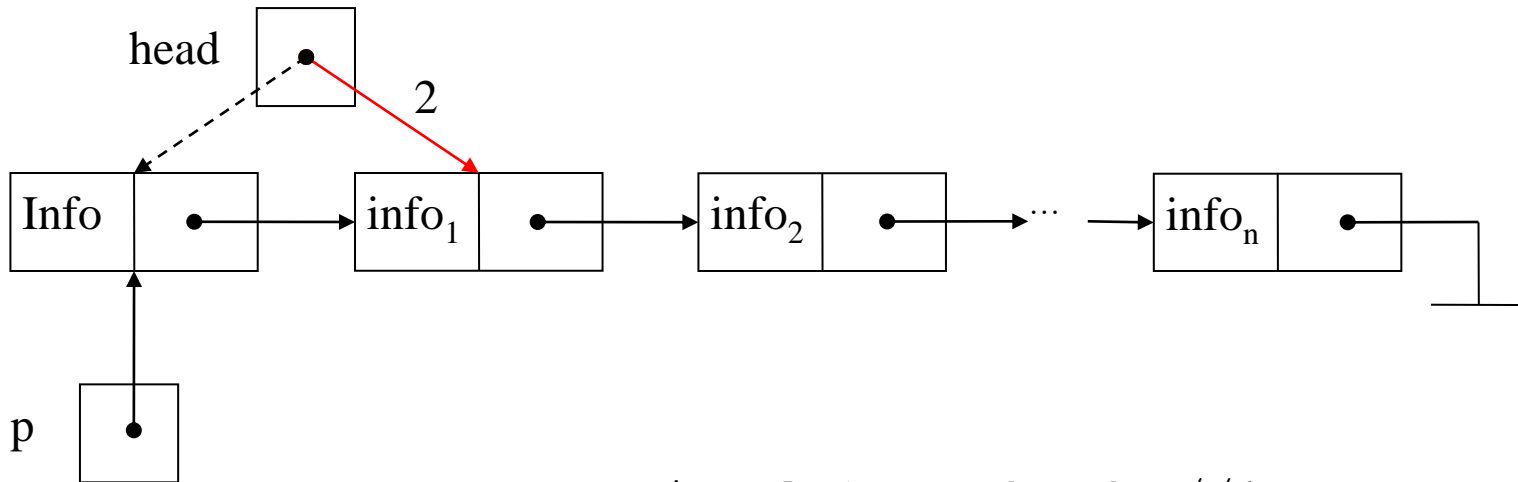
```
ListElm* p = head; //1
```



# Односвязные списки.

## Удаление элемента из головы списка

ШАГ 2

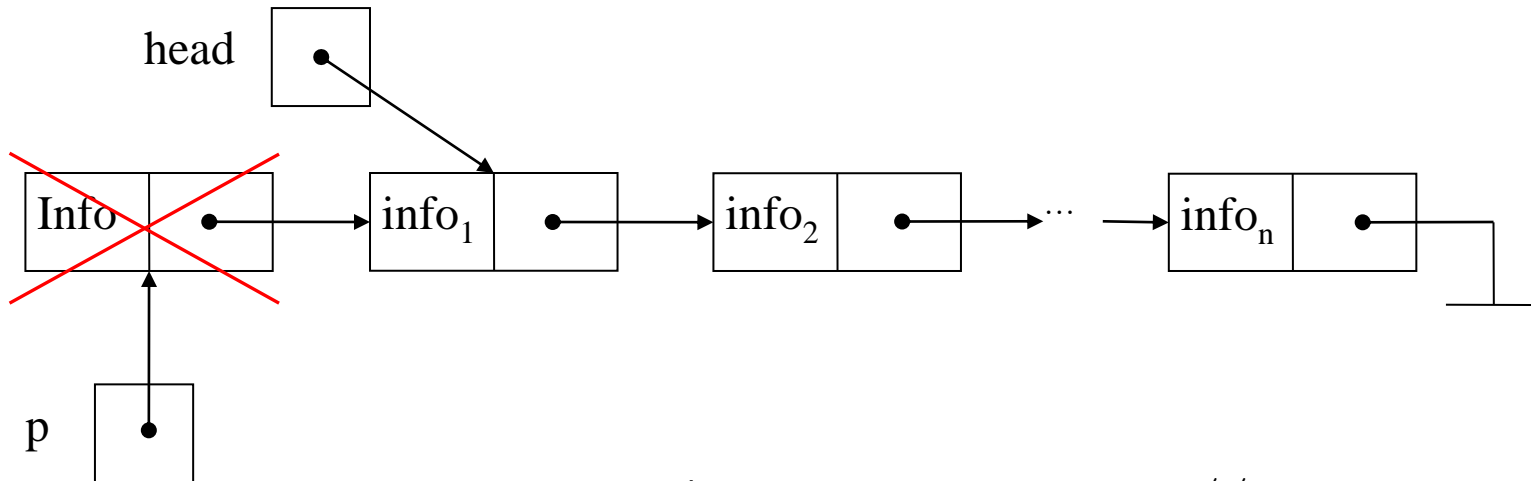


```
ListElm* p = head; //1  
head = p->link;    //2  
...
```

# Односвязные списки.

## Удаление элемента из головы списка

ШАГ 3



```
ListElm* p = head; //1  
head = p->link;    //2  
delete p;        //3  
p = NULL;
```

# Поиск элемента в списке по информационному полю

```
ListElm* p = head;    // 1  
while (p && (p->info != Info))    p = p->link;    //2
```

По выходу из цикла указатель p будет содержать адрес искомого элемента.  
В том случае, если элемент не будет найден, указатель будет нулевым.

Помимо приведенных выше операций с односвязными списками также часто используются такие операции, как добавление и удаление элементов, следующих за указанным. Реализовать такие операции не представляет никаких трудностей. Однако, например, вставку элемента перед указанным или удаление указанного элемента выполнить уже сложнее, так как для этого необходимо получить указатель на предыдущий элемент списка.

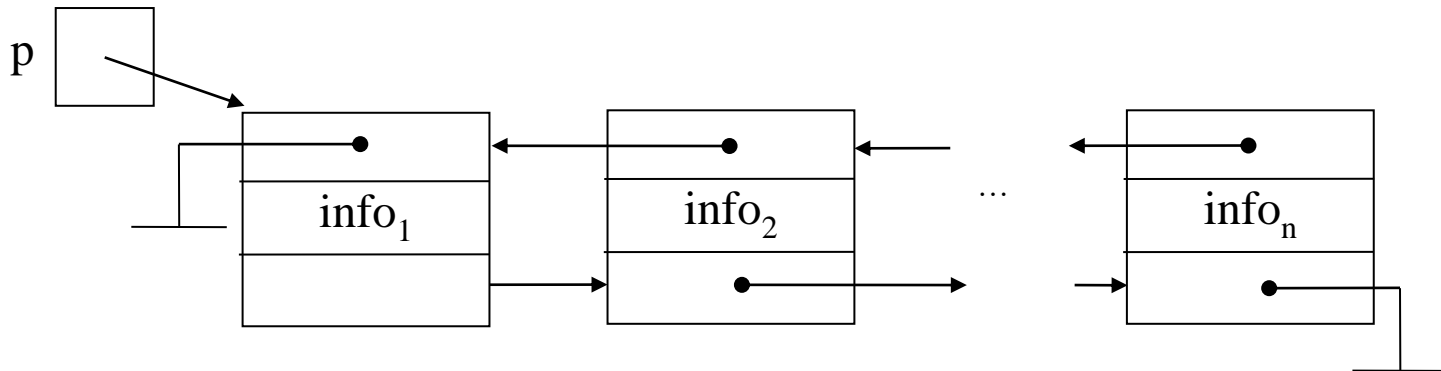
# Двусвязные списки

Каждый элемент такого списка содержит два поля связи, связывающих его с предыдущим и последующим элементами.

```
struct DListElm{  
    T info;  
    ListElm *prev, *next;  
};
```

В связи с тем, что в таких списках от любого элемента можно получить доступ, как к предыдущему, так и последующему элементу, некоторые операции выполняются проще, чем в односвязном списке.

В частности, облегчается выполнение таких операций, как удаление элемента, на котором установлен указатель, и вставка элемента перед элементом, на котором установлен указатель.



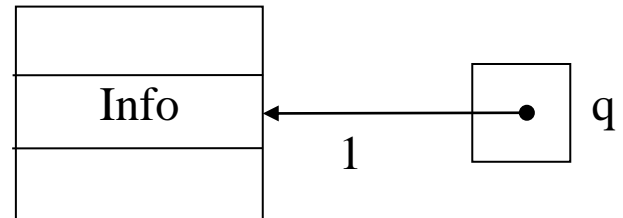
# Двусвязные списки. Вставка элемента

Исходное состояние



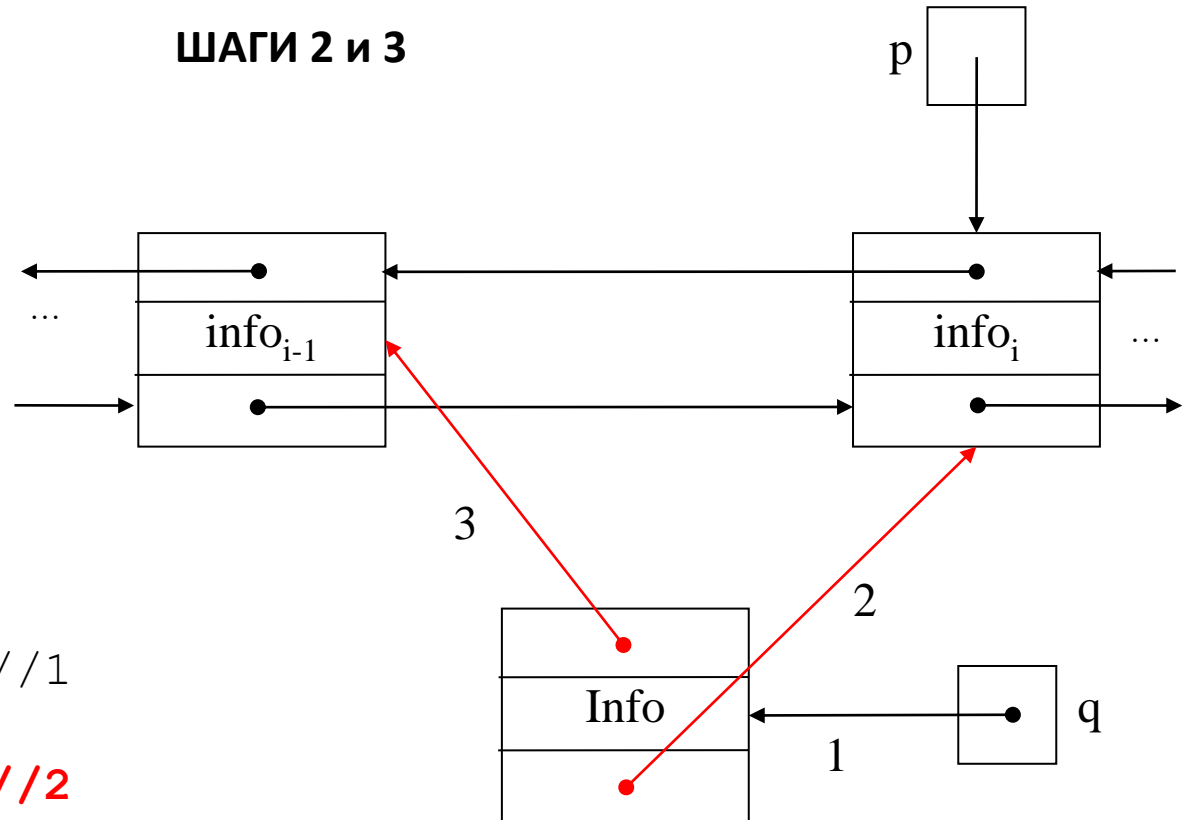
ШАГ 1

```
DListElm* q =  
    new DListElm; //1  
q->info = Info;  
...
```



# Двусвязные списки. Вставка элемента

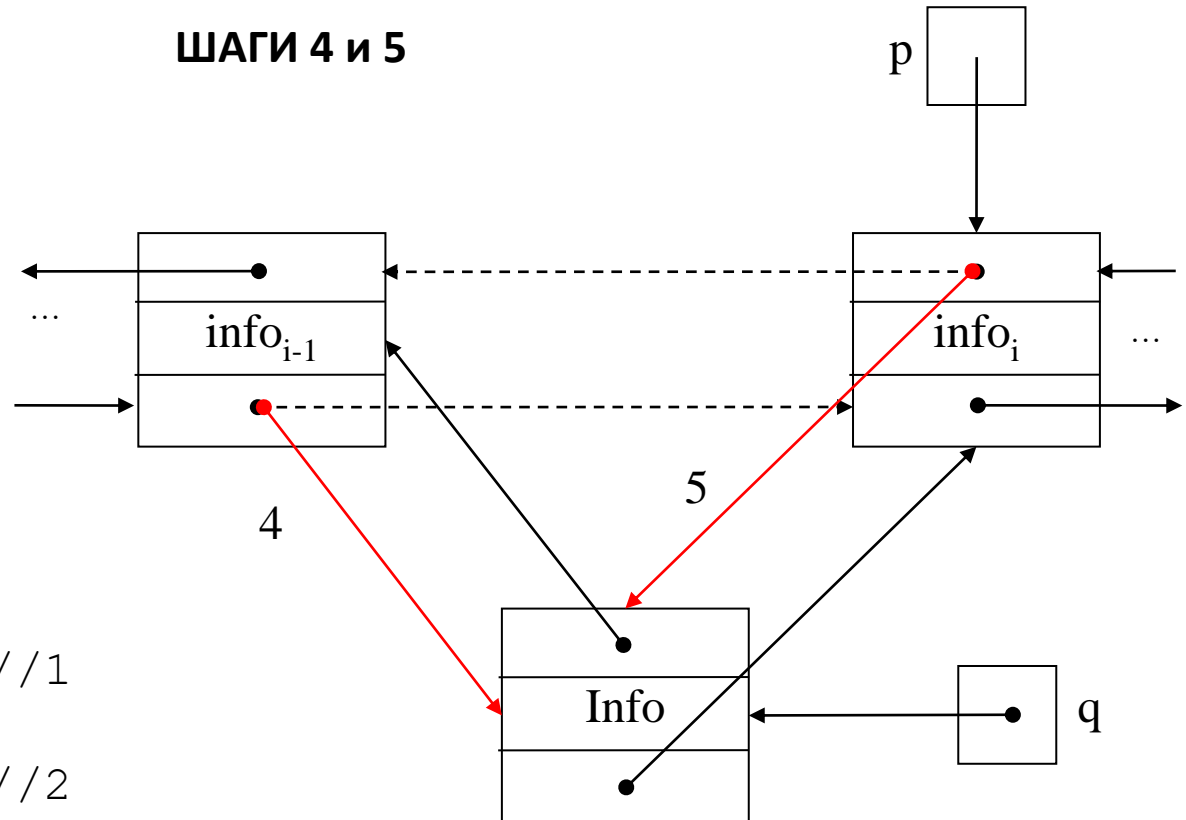
ШАГИ 2 и 3



```
DListElm* q =  
    new DListElm; //1  
q->info = Info;  
q->next = p; //2  
q->prev = p->prev; //3  
...
```

# Двусвязные списки. Вставка элемента

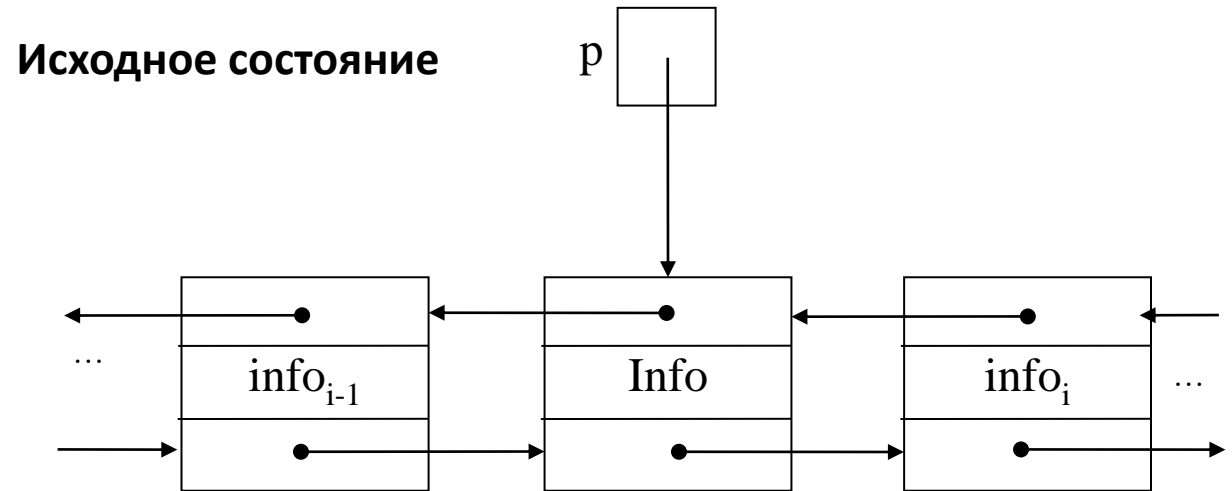
ШАГИ 4 и 5



```
DListElm* q =  
    new DListElm; //1  
q->info = Info;  
q->next = p;      //2  
q->prev = p->prev; //3  
p->prev->next = q; //4  
p->prev = q;      //5
```

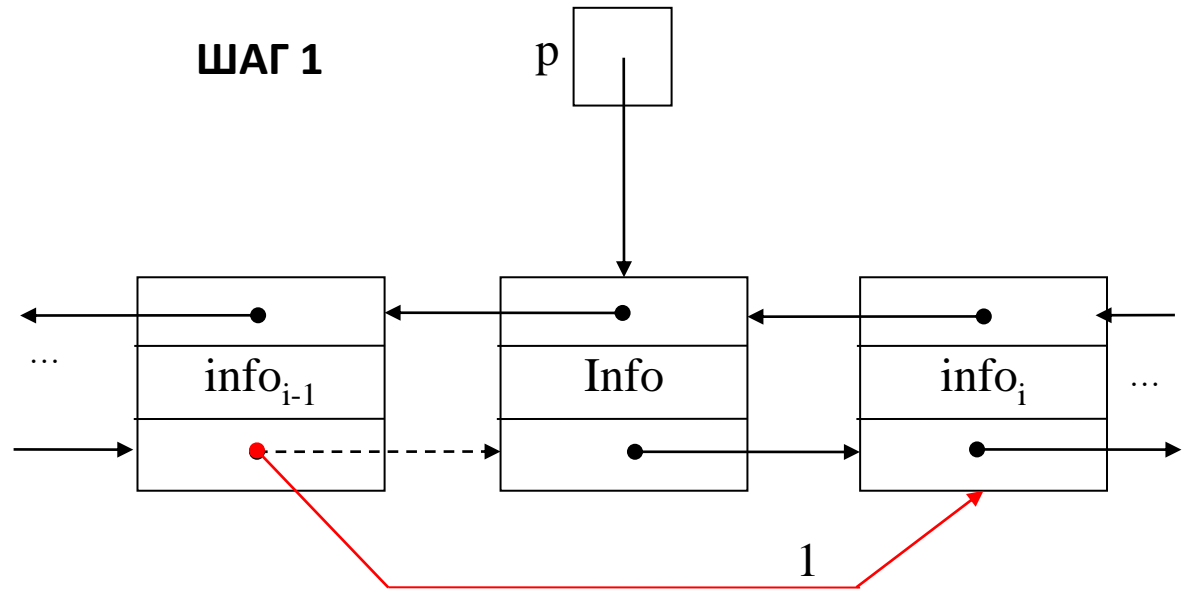
Приведенный код вставки элемента будет приводить к ошибке в том случае, если  $q$  указывает на головной элемент списка.

# Двусвязные списки. Удаление элемента





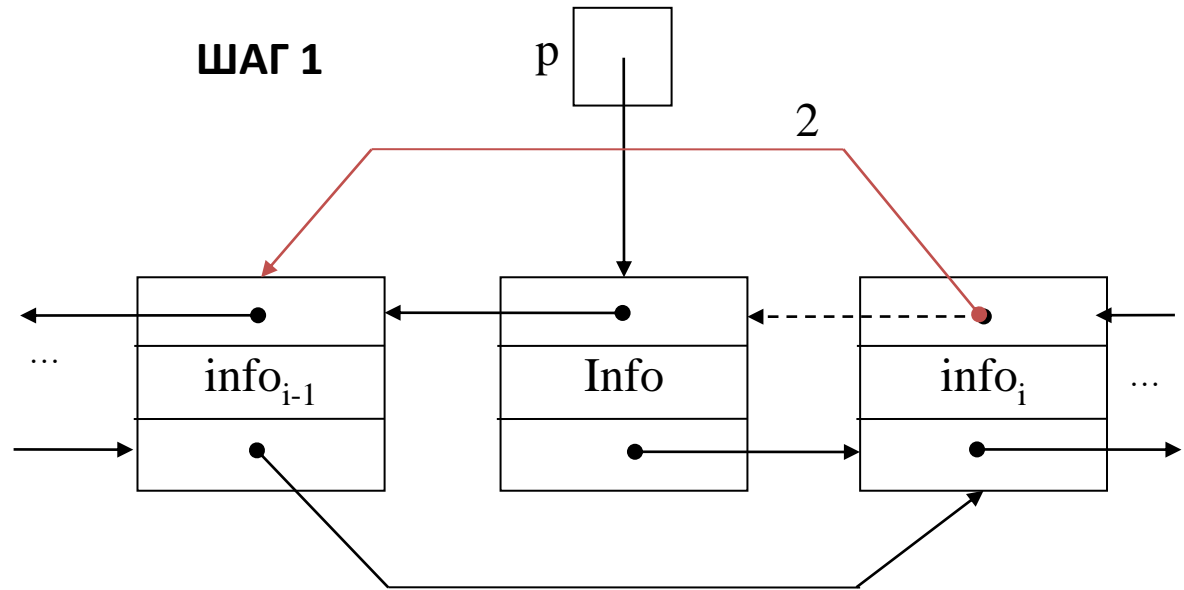
# Двусвязные списки. Удаление элемента



```
p->prev->next = p->next; //1
```

...

# Двусвязные списки. Удаление элемента

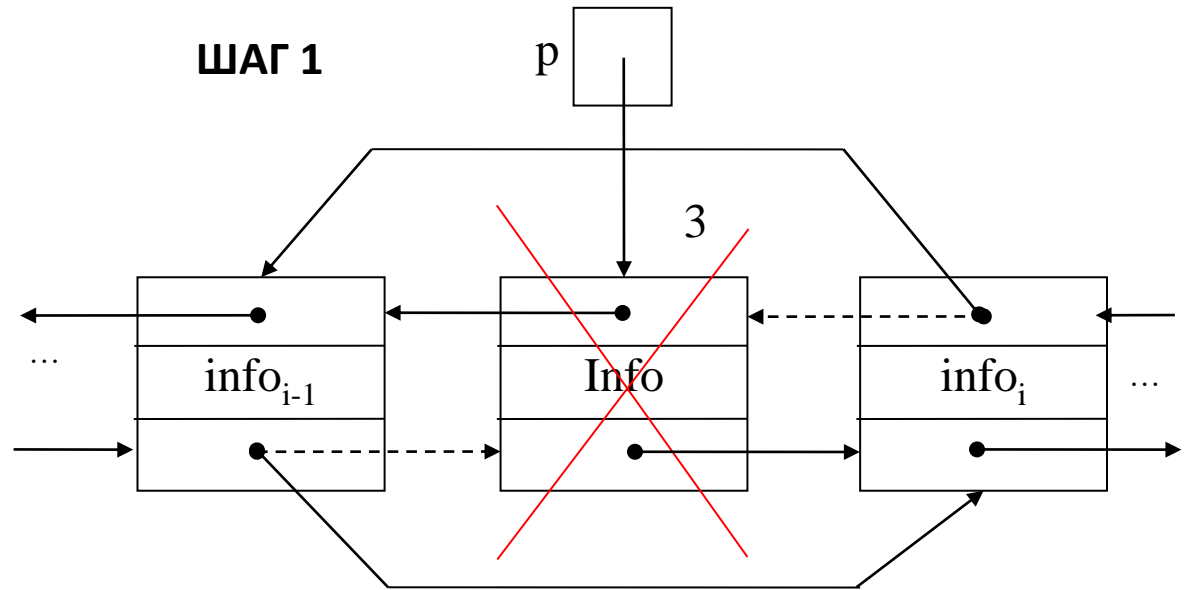


```
p->prev->next = p->next; //1
```

```
p->next->prev = p->prev; //2
```

```
...
```

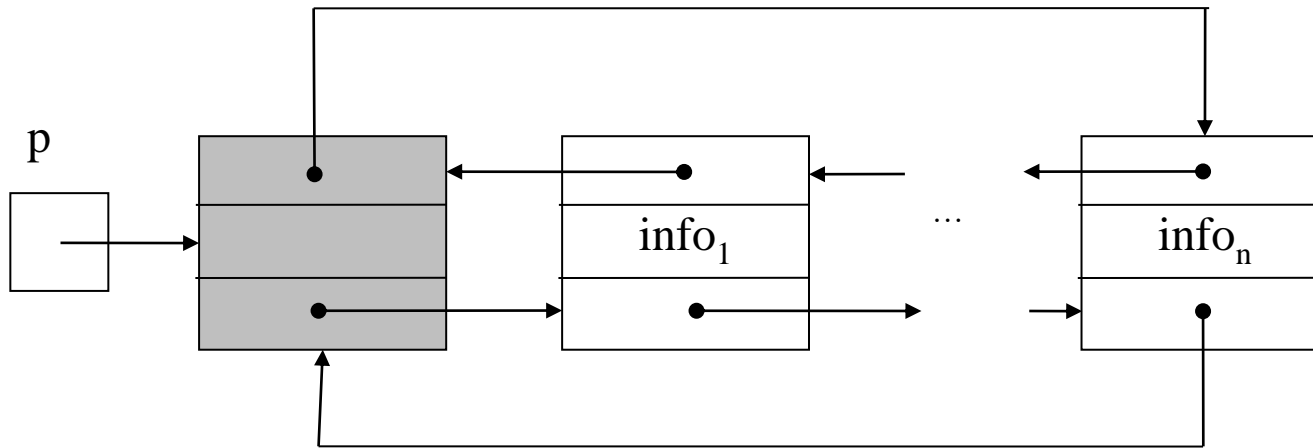
# Двусвязные списки. Удаление элемента



```
p->prev->next = p->next; //1
p->next->prev = p->prev; //2
delete p;           //3
p = NULL;
```

Код удаления элемента будет приводить к ошибкам, как для головного, так и для концевых элементов списка. Поэтому такие случаи должны обрабатываться отдельно.

# Циклические списки



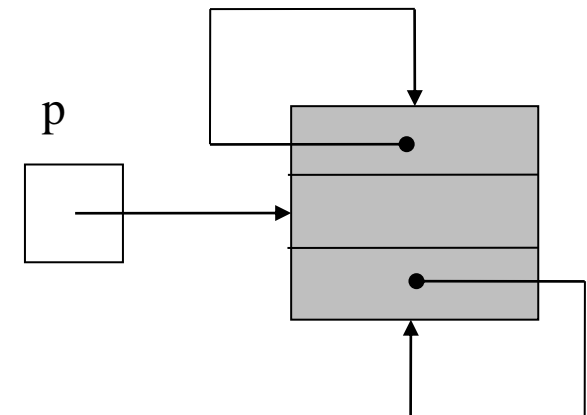
Избавиться от проблем с головным и концевым элементами помогает использование циклических списков.

В таких списках вводится **специальный головной элемент**, который создается при создании списка и не удаляется. Его поле связи с предыдущим элементом содержит указатель на концевой элемент списка, а поле связи концевого элемента со следующим содержит указатель на головной элемент.

Когда список пуст, поля связи головного элемента указывают на сам этот элемент.

Наличие выделенного головного элемента позволяет различить ситуации «список не существует» и «список пуст».

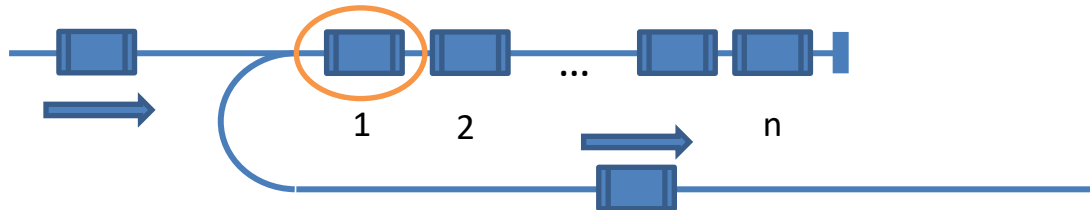
Работа со всеми элементами циклического списка осуществляется единообразно



# Виды списков

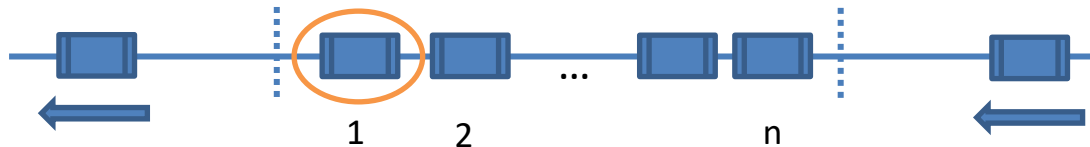
**Стек** – включение, исключение и доступ к элементам производится на одном конце списка, называемом вершиной стека.

*LIFO* – “последним пришел – первым вышел”.

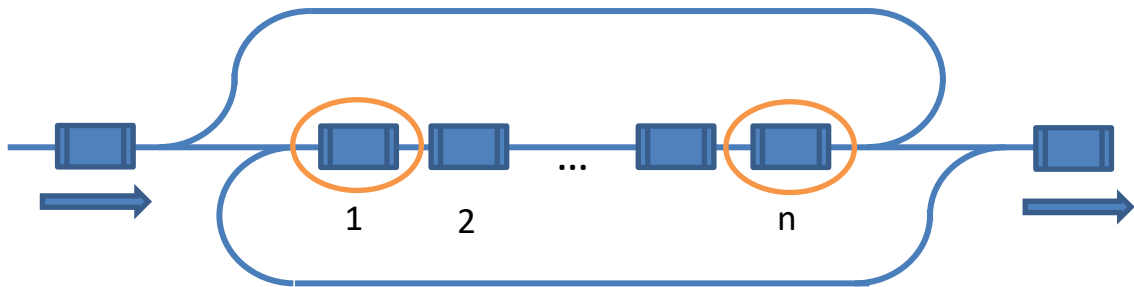


**Очередь** – включение элемента производится в хвост, а исключение и доступ к элементам - в начале списка.

*FIFO* – “первым пришел – первым вышел”



**Дек** (двусторонняя очередь) –включение, исключение и доступ к элементам производится как в начале, так и в хвосте списка.

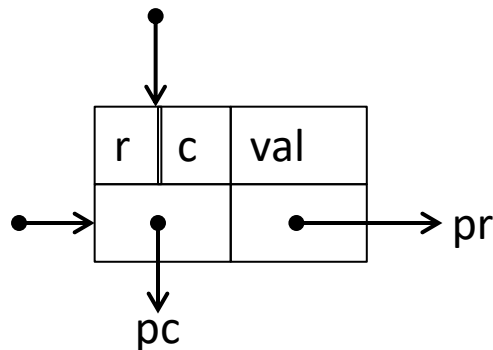


# Виды списков

**Разнородным** называется список, включающий элементы различных типов. В каждый элемент вводится дополнительный атрибут, определяющий тип элемента списка и соответствующим образом интерпретировать содержимое его информационных полей

**Ортогональный список (мультисписок)** – это структура, каждый элемент которой входит более чем в один список.

Каждый элемент ортогонального списка имеет несколько полей, в соответствии с количеством списков (все операции выполняются для каждого списка).



**r** – номер строки

**c** – номер столбца

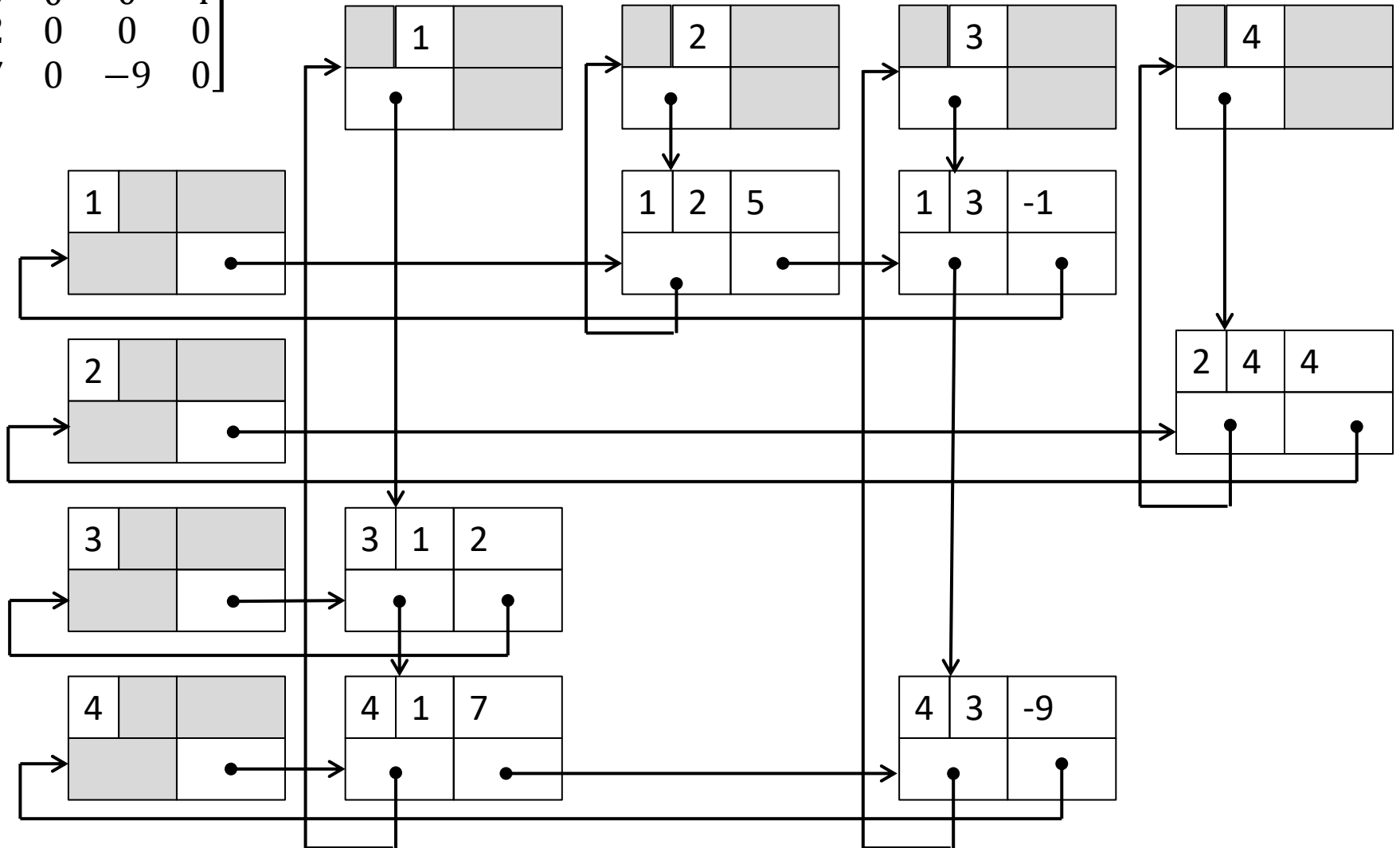
**val** – значение

**pc** – следующий по столбцу

**pr** – следующий по строке

# Ортогональные списки для разреженных матриц

$$\begin{bmatrix} 0 & 5 & -1 & 0 \\ 0 & 0 & 0 & 4 \\ 2 & 0 & 0 & 0 \\ 7 & 0 & -9 & 0 \end{bmatrix}$$



**STL**



# Последовательные контейнеры

**vector<T,A>** вектор - последовательно расположенный набор элементов T  
**array<T, N>** вектор постоянной длины - последовательно расположенный набор элементов T (C++ 11)  
**list<T,A>** двусвязный список элементов T  
**forward\_list<T,A>** односвязный список элементов T (C++ 11)  
**deque<T,A>** дек – двусторонняя очередь элементов T

## Общие характеристики

### Контейнер

вектор  
списки

### Преимущества

обращение по индексу  
вставка/удаление

### Недостатки

вставка/удаление  
доступ к произвольному эл-ту

## Параметры:

T – тип хранимых элементов

A – аллокатор, определяющий способ размещения и освобождения памяти

По-умолчанию аллокатор **A = std::allocator<T>**:

выделение - **operator new()**

освобождение - **operator delete()**

# Вектор (vector)

**vector<T,A>** вектор - последовательно расположенный набор элементов T

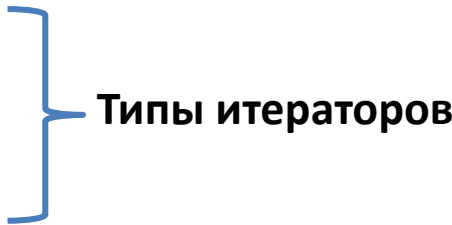
## Особенности:

- последовательное хранение (указатель на элемент вектора может передаваться в функции)
- произвольный доступ к элементам
- возможность изменения длины (автоматическое перераспределение памяти происходит не при каждой операции)
- быстрая вставка и удаление в конце
- частичное сохранение корректности итераторов (инвалидация)

# Вектор (vector)

## Внутренние типы:

```
template<typename T, typename Allocator = allocator<T>>
class vector {
public:
    using iterator = ...
    using const_iterator = ...;
    using reverse_iterator = ...;
    using const_reverse_iterator = ...;
    using value_type = T;           // Тип значения эл-та
    using pointer = ...;            // Тип указателя на эл-т
    using reference = ...;          // Тип ссылки на эл-т
    using const_reference = ...;    // Тип константной ссылки на эл-т
    using const_pointer = ...;      // Тип константного ук-ля на эл-т
    using difference_type = ...;    // Тип результата вычитания итераторов
    using size_type = ...;          // Тип размера
    using allocator_type = Allocator; // Тип аллокатора
    // ...
};
```



Типы итераторов

# Вектор (vector)

## Создание вектора:

```
vector<int> v1;           // создание вектора с конструктором по-умолчанию

vector<int> v1(100);      // создание вектора из 100 эл-тов
vector<T>  vt1(100);      // ... с инициализацией эл-тов по-умолч. T::T()

vector<int> v2(100, 3);   // создание вектора из 100 эл-тов
vector<T>  vt2(100, v);   // ... с инициализацией эл-тов к.к. T::T(T&)

vector<int> v3(v2);       // создание вектора с исп к.к.
vector<T>  vt3(vt2);

vector<T>  vt4(iter_l, iter_r); // создание вектора, содержащего эл-ты,
                                // начиная с iter_l по iter_r
```

# Вектор (vector)

## Доступ к элементам вектора:

<code>reference at( size_type pos );</code>	<code>// предоставляет доступ к указанному</code>
<code>const_reference at( size_type pos ) const;</code>	<code>// элементу с проверкой границ</code>
<code>reference operator[]( size_type pos );</code>	<code>// предоставляет доступ к указанному</code>
<code>const_reference operator[]( size_type pos ) const;</code>	<code>// элементу</code>
<code>reference front();</code>	<code>// предоставляет доступ к первому</code>
<code>const_reference front() const;</code>	<code>// элементу</code>
<code>reference back();</code>	<code>// предоставляет доступ к последнему</code>
<code>const_reference back() const;</code>	<code>// элементу</code>
<code>T* data() noexcept;</code>	<code>// прямой доступ к базовому массиву</code>
<code>const T* data() const noexcept;</code>	

# Вектор (vector)

## Добавление элементов вектора:

```
iterator insert( const_iterator pos,  
                const T& value );  
iterator insert( const_iterator pos,  
                size_type count, const T& value );  
iterator insert( const_iterator pos,  
                InputIt first, InputIt last );  
iterator insert( const_iterator pos,  
                std::initializer_list<T> ilist );
```

```
iterator emplace( const_iterator pos,  
                 Args&&... args );
```

```
void push_back( const T& value );
```

```
void emplace_back( Args&&... args );
```

```
// вставка перед элементом, на  
// который указывает pos  
// вставка count копий value перед  
// элементом  
// вставка элементов из диапазона  
// [first, last) перед элементом  
// вставка элементов из списка ilist
```

```
// вставка элемента с конструиро-  
// ванием его на месте по args
```

```
// добавление элемента в конец
```

```
// добавление в конец с  
// конструированием
```

# Вектор (vector)

## Удаление элементов вектора:

`void clear();`

**// очистка**

`iterator erase( const_iterator pos );`  
`iterator erase( const_iterator first,`  
`const_iterator last );`

**// удаление элемента в позиции**

**// удаление элементов в диапазоне**

`void pop_back();`

**// удаление последнего элемента**

`void resize( size_type count );`  
`void resize( size_type count,`  
`const value_type& value);`

**// изменение размера с усечением**  
**или дополнением**

`void swap( vector& other );`

**// обмен значениями с другим**  
**вектором**

# Вектор (vector)

## Другие функции:

<code>empty</code>	<code>// проверяет, пуст ли контейнер</code>
<code>size</code>	<code>// возвращает количество элементов</code>
<code>max_size</code>	<code>// возвращает максимально возможное количество элементов</code>
<code>reserve</code>	<code>// резервирует память</code>
<code>capacity</code>	<code>// возвращает количество элементов, которые могут храниться в выделенной в данный момент памяти</code>
<code>shrink_to_fit (C++11)</code>	<code>// уменьшает использование памяти за счёт освобождения неиспользуемой памяти</code>



# Вектор (vector)

## Итераторы:

```
iterator begin() noexcept;  
const_iterator begin() const noexcept;  
const_iterator cbegin() const noexcept;
```

**// итератор на первый элемент**

```
iterator end() noexcept;  
...  
const_iterator cend() const noexcept;
```

**// итератор на элемент, следующий  
за последним элементом**

```
reverse_iterator rbegin() noexcept;  
...  
const_reverse_iterator crbegin() const noexcept;
```

**// обратный итератор на первый  
элемент перевёрнутого vector**

```
reverse_iterator rend() noexcept;  
...  
const_reverse_iterator crend() const noexcept;
```

**// обратный итератор на элемент,  
следующий за последним  
элементом перевёрнутого vector**

# Вектор (vector)

## Пример:

```
#include <vector>

int main()
{
    std::vector<int> nums {1, 2, 3, 4, 5};

    std::cout << " a[2] = " << nums[2] << '\n';    // 3
    nums[2] = 1;

    nums. push_back (6);

    for (auto v : nums) {
        std::cout << ' ' << v;                      // 1 2 1 4 5 6
    }
}
```

# Список (list)

**list<T,A>**                    двусвязный список элементов T

## **Особенности:**

- доступ к элементам через двунаправленные итераторы
- переменная длина
- быстрая вставка и удаление в любой позиции
- гарантируется сохранение корректности итераторов после вставки и удаления (кроме самих удаляемых эл-тов)

**Набор встроенных типов, порядок создания** – аналогично вектору

**Собственный набор операций** для списка

(ряд стандартных алгоритмов невозможно применить ):

sort, unique, merge, reverse, remove, remove\_if

# Список (list)

## Вставка элементов:

```
iterator insert( const_iterator pos,  
                 const T& value );  
iterator insert( const_iterator pos,  
                 size_type count, const T& value );  
iterator insert( const_iterator pos,  
                 InputIt first, InputIt last );  
iterator insert( const_iterator pos,  
                 std::initializer_list<T> ilist );
```

```
iterator emplace( const_iterator pos,  
                  Args&&... args );
```

```
void push_back( const T& value );  
void emplace_back( Args&&... args );
```

```
void push_front( const T& value );  
void emplace_front( Args&&... args )
```

```
// вставка перед элементом, на  
// который указывает pos  
// вставка count копий value перед  
// элементом  
// вставка элементов из диапазона  
// [first, last) перед элементом  
// вставка элементов из списка ilist
```

```
// вставка элемента с конструиро-  
// ванием его на месте по args
```

```
// добавление элемента в конец  
// добавление в конец с  
// конструированием
```

```
// добавление в начало  
// добавление в начало с  
// конструированием
```

# Список(list)

## Удаление элементов :

`void clear();`

`// очистка`

`iterator erase( const_iterator pos );`  
`iterator erase( const_iterator first,`  
`const_iterator last );`

`// удаление элемента в позиции`  
`// удаление элементов в диапазоне`

`void pop_back();`

`// удаление последнего элемента`

`void pop_front();`

`// удаление первого элемента`

## Изменение размера:

`void resize( size_type count );`  
`void resize( size_type count,`  
`const value_type& value);`

`// изменение размера с усечением`  
`или дополнением`

## Обмен:

`void swap( vector& other );`

`// меняет местами содержимое`  
`списков`

# Список (list)

## Другие функции:

<code>empty</code>	<code>// проверяет, пуст ли контейнер</code>
<code>size</code>	<code>// возвращает количество элементов</code>
<code>max_size</code>	<code>// возвращает максимально возможное количество элементов (с учетом ограничений наложенных системой или реализацией)</code>
<del><code>reserve</code></del>	
<del><code>capacity</code></del>	
<del><code>shrink_to_fit (C++11)</code></del>	

# Список (list)

## Операции со списком:

**void merge( list& other );**  
**void merge( list& other, Compare comp );**

**// слияние двух отсортированных списков**

**void splice(const\_iterator pos, list& other);**  
**void splice(const\_iterator pos, list& other, const\_iterator it);**  
**void splice(const\_iterator pos, list& other, const\_iterator first, const\_iterator last);**

**// перемещает элементы из другого списка**

**void remove( const T& value );**  
**void remove\_if( UnaryPredicate p );**

**// удаляет элементы, удовлетворяющие определенным критериям**

**void reverse();**

**// инвертирует порядок элементов**

**void unique();**  
**void unique( BinaryPredicate p );**

**// удаляются последовательно повторяющиеся элементы**

**void sort();**  
**void sort( Compare comp );**

**// сортирует элементы**

# Список (list)

## Итераторы:

`iterator begin() noexcept;`

`...`

`const_iterator cbegin() const noexcept;`

**// итератор на первый элемент**

`iterator end() noexcept;`

`...`

`const_iterator cend() const noexcept;`

**// итератор на элемент, следующий  
за последним элементом**

`reverse_iterator rbegin() noexcept;`

`...`

`const_reverse_iterator crbegin() const noexcept;`

**// обратный итератор на первый  
элемент перевёрнутого list**

`reverse_iterator rend() noexcept;`

`...`

`const_reverse_iterator crend() const noexcept;`

**// обратный итератор на элемент,  
следующий за последним  
элементом перевёрнутого list**



# Список (list)

## Пример:

```
#include <list>
```

```
int main()
```

```
{
```

```
    std::list<int> nums;
```

```
    nums.push_back(4);
```

```
    nums.push_back(7);
```

```
    nums.push_back(3);
```

```
    std::list<int> nums2 = nums;
```

```
    nums.sort();
```

```
    nums.splice(nums.end(), nums2);    // перемещает элементы из другого списка
```

```
    for (int i : nums) {
```

```
        std::cout << i << '\n'; // 3 4 7 4 7 3
```

```
    }
```

```
    return 0;
```

```
}
```

# Дек (deque)

**deque<T,A>**      дек – двусторонняя очередь элементов T

## **Особенности:**

- произвольный доступ к элементам
- переменная длина
- быстрая вставка и удаление в начале и в конце
- без гарантий сохранения корректности итераторов после вставки и удаления (итераторы внутрь сохраняются при операциях на концах)

**Набор встроенных типов, порядок создания** – аналогично вектору

# Оценки времени выполнения операций (1)

## Распространенные оценки:

Константное время –  $O(1)$

Линейное время -  $O(N)$

Логарифмическое -  $O(\log(N))$

$N$  логарифм  $N$  –  $O(N \log(N))$

Квадратичное время –  $O(N^2)$

**Задача:** найти минимум в упорядоченном массиве

**Решение:**

```
double* min( double *a, size_t count )  
{  
    if (count) return &(a[0]);  
    else return nullptr;  
}
```

**Оценка времени:** ?

# Оценки времени выполнения операций (1)

## Распространенные оценки:

Константное время –  $O(1)$

Линейное время -  $O(N)$

Логарифмическое -  $O(\log(N))$

$N$  логарифм  $N$  –  $O(N \log(N))$

Квадратичное время –  $O(N^2)$

**Задача:** найти минимум в упорядоченном массиве

**Решение:**

```
double* min( double *a, size_t count )
{
    if (count) return &(a[0]);
    else return nullptr;
}
```

**Оценка времени:**

$O(1)$

# Оценки времени выполнения операций (2)

**Задача:** найти заданное значение в неупорядоченном массиве

**Решение:**

```
int* find( int *a, size_t count , int v )
{
    for (size_t i = 0; i < count; i++)
        if (a[i] == v) return (a+i);
    return nullptr;
}
```

**Оценка времени: ?**

# Оценки времени выполнения операций (2)

**Задача:** найти заданное значение в неупорядоченном массиве

**Решение:**

```
int* find( int *a, size_t count , int v )
{
    for (size_t i = 0; i < count; i++)
        if (a[i] == v) return (a+i);
    return nullptr;
}
```

**Оценка времени:**

$O(N)$  – худший случай

в среднем ?

в лучшем случае ?

# Оценки времени выполнения операций (3)

**Задача:** найти минимум в неупорядоченном массиве

**Решение:**

```
double* min( double *a, size_t count )
{
    if (! count) return nullptr;
    size_t* imin = 0;
    for (size_t i = 1; i < count; i++)
        if (a[i] < a[imin]) imin = i;
    return &(a[imin]);
}
```

**Оценка времени:** ?

# Оценки времени выполнения операций (3)

**Задача:** найти минимум в неупорядоченном массиве

**Решение:**

```
double* min( double *a, size_t count )
{
    if (! count) return nullptr;
    size_t* imin = 0;
    for (size_t i = 1; i < count; i++)
        if (a[i] < a[imin]) imin = i;
    return &(a[imin]);
}
```

**Оценка времени:**

$O(N)$



# Оценки времени выполнения операций (5)

Оценки даны для **наихудшего** случая

Вид операции	Вектор	Список	Дек
Доступ к элементу	$O(1)$	$O(N)$	$O(1)$
Добавление / удаление в начале	$O(N)$	$O(1)$	Амортизированное $O(1)$
Добавление / удаление в середине	$O(N)$	$O(1)$	$O(N)$
Добавление / удаление в конце	Амортизированное $O(1)$	$O(1)$	Амортизированное $O(1)$