

Лекция 12

СПИСКИ

СТРУКТУРЫ ДАННЫХ

- Значения стандартных типов данных можно группировать и создавать *структуры данных*
- Структура данных представляется одной переменной – *именем структуры данных*, а входящие в нее значения – *элементы*, выделяются тем или иным способом, специфичным для каждой такой структуры
- Наиболее простой и часто используемой структурой данных является *массив*

МАССИВЫ

- Массив – это набор некоторого числа однотипных данных, расположенных в последовательных ячейках памяти
- Количество элементов массива называется его *размером*, а тип элементов – *типом массива*

5	7	34	0	11	-6	19	3	11
---	---	----	---	----	----	----	---	----

ПРОБЛЕМЫ РАБОТЫ С МАССИВАМИ

- Первым недостатком массивов является их фиксированный размер, который устанавливается при создании массива и в дальнейшем не может быть изменен
- Частично эта проблема решается при использовании динамических массивов путем создания новых массивов большего размера

СОЗДАНИЕ МАССИВА

- Статический массив. Располагается в статической или автоматической памяти

```
using namespace std;
```

```
const int n = 10;
```

```
int mas[n] = {5, 7, 24, -10, 9, 14, 18, -2, 2, 4};
```

```
void sort (int[ ] a, int length)
```

```
{
```

```
    int j, x;
```

```
    for (int i = 0; i < length; i++){
```

```
        ...
```

```
    }
```

```
}
```

СОЗДАНИЕ МАССИВА

- Динамический массив. Располагается в динамической памяти

```
int *mas, n;  
int _tmain (int argc, _TCHAR* argv[])  
{  
    ...  
    cout << "Задайте размер массива" << endl;  
    cin >> n;  
    mas = new int [n] {4, 7 ,9};  
    ...  
    delete [] mas;  
    ...  
}
```

ПРОБЛЕМЫ РАБОТЫ С МАССИВАМИ

- Второй недостаток массивов связан с тем, что элементы массива занимают смежные ячейки памяти
- Это сильно усложняет выполнение операций добавления и удаления элементов в заполненной части массива

СТРУКТУРЫ

- Еще одним примером структур данных являются *структуры*

struct

```
{   char fio [30];  
  int age, code;  
  double salary;  
} smith;
```

- Здесь объявлена структура с именем smith, содержащая 4 поля

СТРУКТУРЫ

- Структуры также, как и массивы, имеют фиксированный размер, определяемый как сумма размеров их полей
- В отличие от массивов, операции добавления и удаления элементов для структуры невозможны

СТАТИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

- В силу перечисленных особенностей массивы и структуры называют *статическими структурами данных*

ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

- Недостатков статических структур данных лишены структуры данных с изменяющимися во время выполнения программы составом и размерами, называемые *динамическими структурами данных*

ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

- Переменные, входящие в состав динамических структур, необходимо каким-либо образом связывать друг с другом
- Поэтому каждый элемент динамической структуры должен содержать один или несколько адресов связанных с ним элементов, т.е. *указателей* на эти элементы

ЛИНЕЙНЫЕ (ОДНОНАПРАВЛЕННЫЕ) СПИСКИ

- Самый простой способ соединить отдельные элементы между собой заключается в том, чтобы снабдить каждый из них только одним указателем на другой элемент
- В результате получается динамическая структура, называемая *линейным (однонаправленным) списком*

ЛИНЕЙНЫЕ СПИСКИ

- Между элементами линейного списка существует отношение *предыдущий-последующий*

ТИП ДАННЫХ ЭЛЕМЕНТА СПИСКА

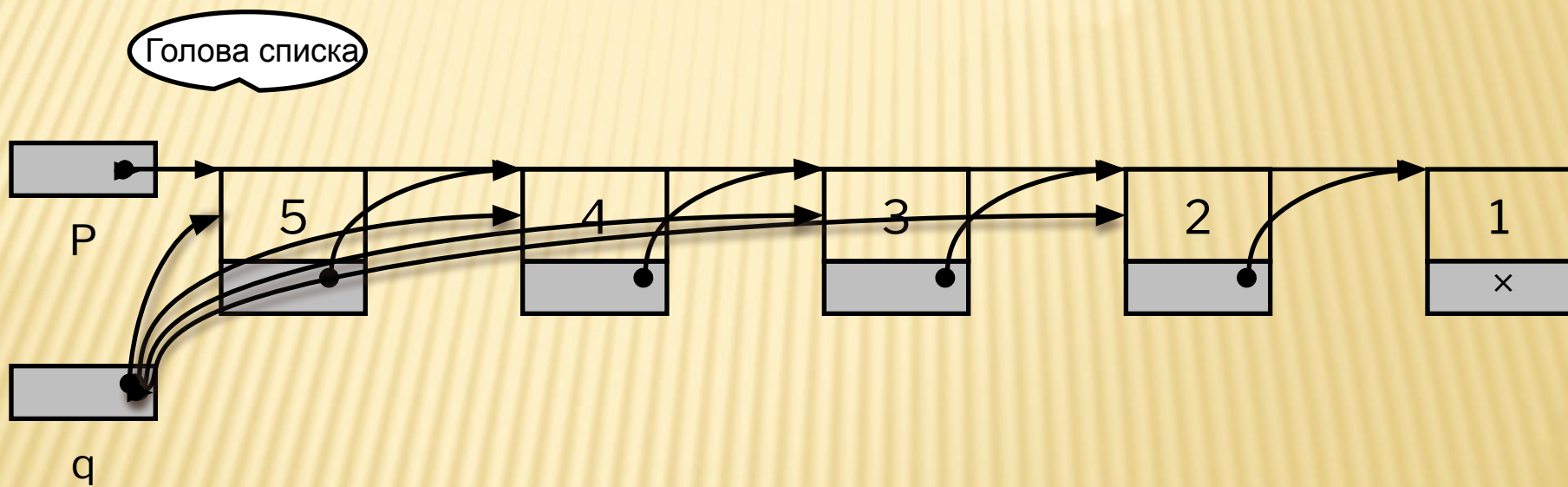
- Для элемента линейного списка можно определить следующий тип данных:

```
struct element
```

```
{ int info;    // информационное поле  
  element* next; // указатель на следующий элемент  
};
```

- Для информационного поля может быть выбран любой другой тип данных, в том числе, массив или структура

ЛИНЕЙНЫЙ СПИСОК



ОПЕРАЦИИ НАД СПИСКАМИ

- Основными операциями при работе со списками являются:
 - инициализация списка
 - проверка списка на пустоту
 - добавление элемента в список
 - удаление элемента из списка
 - поиск в списке

ИНИЦИАЛИЗАЦИЯ СПИСКА

- Эта операция сводится к созданию пустого списка

`p = NULL;`

ПРОВЕРКА СПИСКА НА ПУСТОТУ

- Проверка на пустоту заключается в вычислении значения выражения $p == \text{NULL}$, которое имеет значение `TRUE` в случае, если список пуст, и `FALSE` в противном случае

ДОБАВЛЕНИЕ ЭЛЕМЕНТА В ПУСТОЙ СПИСОК

- Операция сводится к созданию нового элемента с помощью указателя на голову списка

```
p = new elem;
```

```
p->next = NULL;
```

```
x = rand() % 100;
```

```
p->info = x;
```

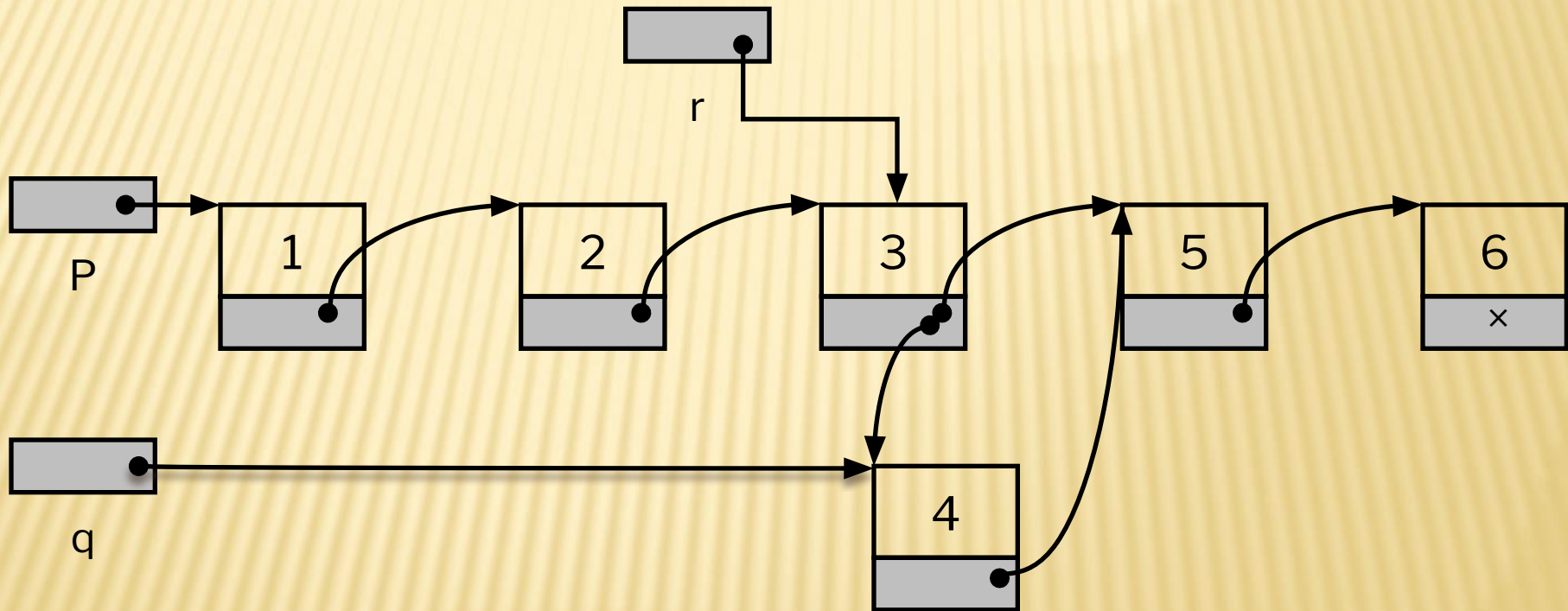
ДОБАВЛЕНИЕ ЭЛЕМЕНТА В НЕПУСТОЙ СПИСОК

- Предполагается, что предварительно в списке тем или иным способом выделен некоторый элемент
- Далее, возможны две ситуации:
 - новый элемент нужно вставить *перед* выделенным;
 - новый элемент нужно вставить *после* выделенного
- Рассмотрим каждую из них в отдельности

ДОБАВЛЕНИЕ ПОСЛЕ ВЫДЕЛЕННОГО

- Для этого необходимо выполнить следующие действия:
 1. определить рабочую переменную-указатель
 2. создать новый элемент с помощью рабочего указателя
 3. связать новый элемент со следующим за выделенным
 4. связать выделенный элемент с новым

ДОБАВЛЕНИЕ ПОСЛЕ ВЫДЕЛЕННОГО



ДОБАВЛЕНИЕ ПОСЛЕ ВЫДЕЛЕННОГО

```
q = new elem;          // создать новый элемент
q->next = r->next;     // связать его со следующим за
                      // выделенным
r->next = q;          // связать выделенный элемент с
                      // новым
x = rand() % 100;
q->info = x;
q = NULL;
```


ДОБАВЛЕНИЕ ПЕРЕД ВЫДЕЛЕННЫМ

- В этом случае задача сводится к предыдущей, а именно, нужно:
 1. добавить новый элемент после выделенного,
 2. произвести обмен значениями между выделенным и новым элементами

ДОБАВЛЕНИЕ ПЕРЕД ВЫДЕЛЕННЫМ

```
q= new elem;      // создать новый элемент
q->next = r->next;  // связать его со следующим за
                    // выделенным
r->next = q;       // связать выделенный элемент с
                    // новым
x = rand() % 100;
q->info = r->info;  // обмен значениями
r->info = x;
```

СПОСОБЫ ВВОДА ДАННЫХ В СПИСОК

- Операции добавления элементов в список могут различаться способом ввода данных
- Данные могут задаваться
 - путем консольного ввода,
 - путем считывания из файла,
 - путем использования генератора случайных чисел

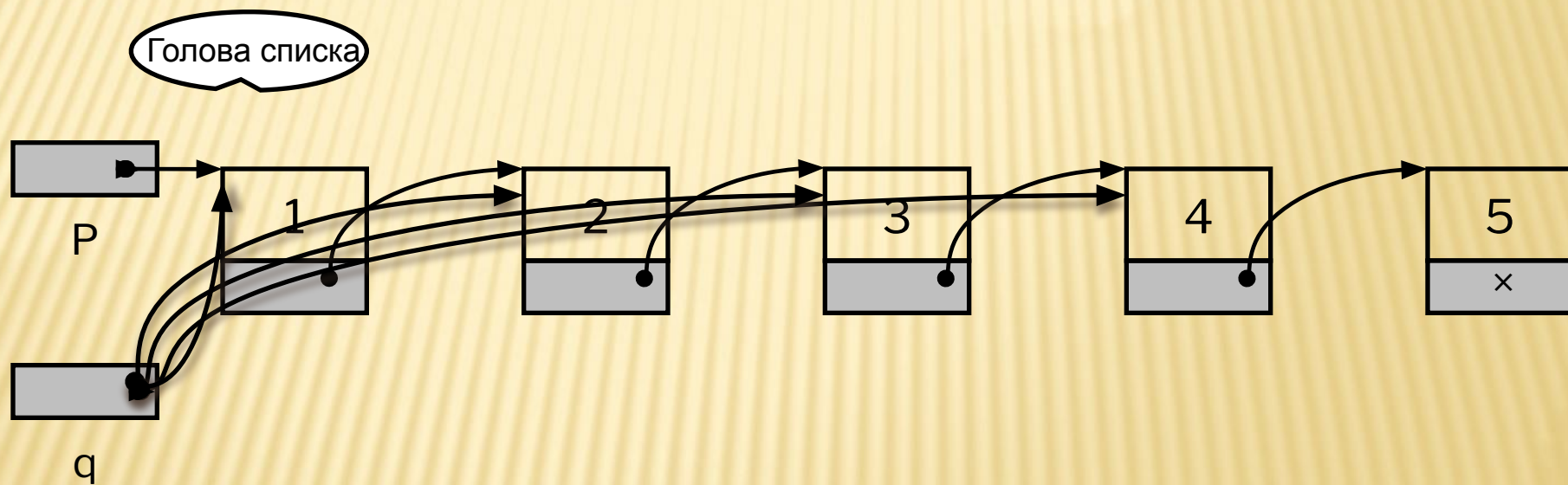
СОЗДАНИЕ СПИСКА

- Операции добавления в список позволяют создавать списки как с прямым, так и с обратным по отношению к порядку ввода следованием элементов
- Алгоритм создания списка:
 1. инициировать список
 2. повторить нужное число раз операцию добавления элемента в список

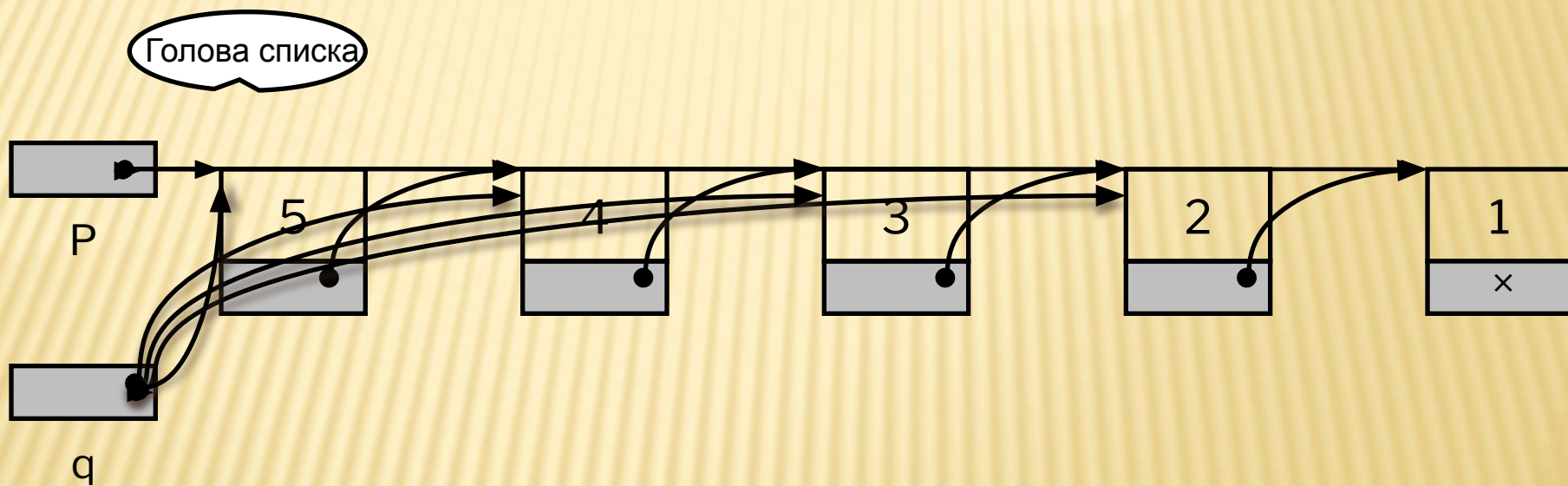
СОЗДАНИЕ СПИСКА

- В зависимости от выбора способа добавления получим прямой или инвертированный список

СОЗДАНИЕ ПРЯМОГО СПИСКА



СОЗДАНИЕ ИНВЕРТИРОВАННОГО СПИСКА



ПОИСК В СПИСКЕ

```
q = p;    //поиск заданного значения x
while (q->next != NULL && q->info!=x)
    q = q->next;
if (q->info==x)
    cout << «Значение найдено»;
else
    cout << «Значение не найдено»;
```


УДАЛЕНИЕ ЭЛЕМЕНТА ИЗ СПИСКА

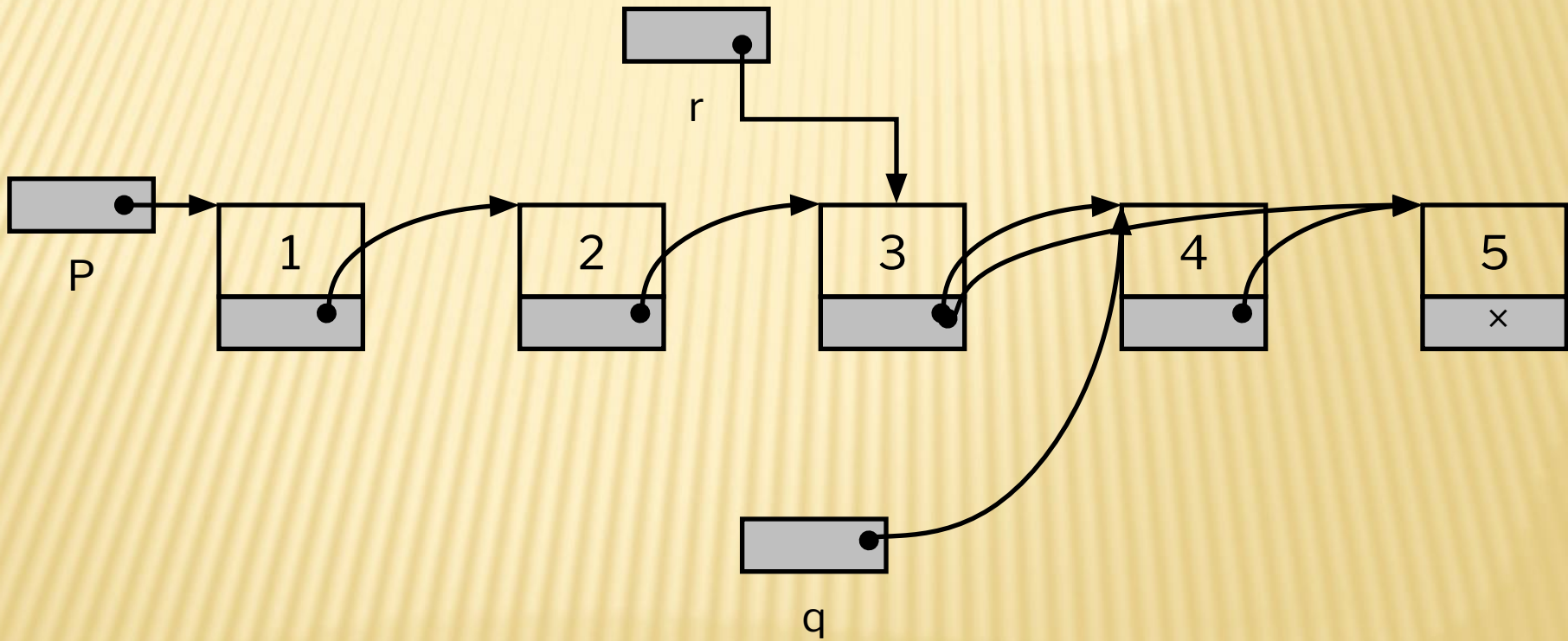
- Особенность этой операции заключается в том, что удалить можно только элемент, *следующий* за выделенным
- Алгоритм удаления состоит просто в изменении значения поля указателя выделенного элемента:

```
q = r->next;
```

```
r->next = q->next;
```

```
delete *q;
```

УДАЛЕНИЕ ЭЛЕМЕНТА ИЗ СПИСКА



УДАЛЕНИЕ ПЕРВОГО ЭЛЕМЕНТА СПИСКА

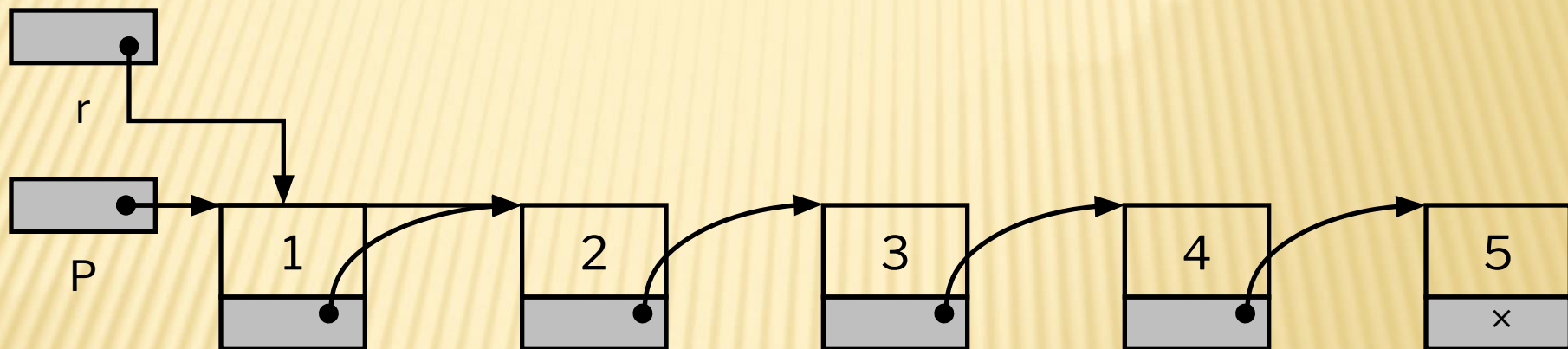
- Особым случаем является удаление первого элемента списка, которое сводится изменению значения указателя на голову списка:

```
p = r->next;
```

```
delete *r;
```

- Разумеется, удаление элемента возможно только при условии, что список не пуст

УДАЛЕНИЕ ПЕРВОГО ЭЛЕМЕНТА СПИСКА



ДОПОЛНИТЕЛЬНО: ПРОСМОТР СПИСКА

- Операция заключается в последовательном переборе всех элементов списка от первого до последнего
- Просмотр списка может сопровождаться выводом значений информационных полей, поиском максимального значения и т.д.
- Операция реализуется простым циклом **for**

ПРИМЕР РЕАЛИЗАЦИИ СПИСКА

- Рассмотрим пример реализации линейного списка в виде динамической структуры
- Тестирование приложения

ДВУНАПРАВЛЕННЫЕ СПИСКИ

- Двунаправленные списки отличаются от однонаправленных тем, что между их элементами существуют отношения *предыдущий-последующий* и *последующий-предыдущий*

ДВУНАПРАВЛЕННЫЕ СПИСКИ

- Небольшое усложнение структуры элемента списка позволяет получить возможность просмотра его в двух направлениях: от начала к концу и от конца к началу

```
struct element
```

```
{ int info; // информационное поле  
  element* prev; // указатель на предыдущий  
  element* next; // указатель на следующий  
};
```


ОПЕРАЦИИ ДОБАВЛЕНИЯ И УДАЛЕНИЯ

- Реализации этих операций в двунаправленных списках имеют свои особенности благодаря возможности доступа к предыдущему и последующему элементам
- Так добавление элемента *перед* выделенным уже не требует обмена значениями и отличается от аналогичной операции добавления *после* выделенного только способом задания значений ссылок

ОПЕРАЦИИ ДОБАВЛЕНИЯ ЭЛЕМЕНТА

□ Добавить *перед*

```
q = new elem;  
q->next = r;  
q->prev = r->prev;  
x = rand() % 100;  
q->info = x;  
r->prev = q;  
r->prev->next = q;  
q = NULL;
```

□ Добавить *после*

```
q = new elem;  
q->prev = r;  
q->next = r->next;  
x = rand() % 100;  
q->info = x;  
r->next = q;  
r->next->prev = q;  
q = NULL;
```

ДОБАВЛЕНИЕ НА КРАЯХ СПИСКА

□ Добавить *первый*

```
q = new elem;  
q->next = p;  
q->prev = NULL;  
x = rand() % 100;  
q->info = x;  
p->prev = q;  
p = q;  
q = NULL;
```

□ Добавить *последний*

```
q = new elem;  
q->prev = r;  
q->next = NULL;  
x = rand() % 100;  
q->info = x;  
r->next = q;  
q = NULL;
```

УДАЛЕНИЕ ЭЛЕМЕНТА

- Удалить *перед*
текущим

```
q=r->prev;  
r->prev = q->prev;  
q->prev->next = r;  
delete *q;
```

- Удалить *после*
текущего

```
q=r->next;  
r->next = q->next;  
q->next->prev = r;  
delete *q;
```

УДАЛЕНИЕ ЭЛЕМЕНТА

- В двунаправленном списке можно удалить и текущий элемент:

```
r->prev->next = r->next;
```

```
r->next->prev = r->prev;
```

```
delete *r;
```

КОЛЬЦЕВЫЕ СПИСКИ

- Кольцевой однонаправленный список получается из линейного «замыканием» последнего элемента на первый
- Соответственно, операция добавления в конец такого списка должна завершаться следующим присваиванием:

`q->next = p;`

КОЛЬЦЕВЫЕ СПИСКИ

- Для двунаправленного кольцевого списка требуется установить две ссылки:
 - первого элемента на последний,
 - последнего элемента на первый
- Ссылка первого элемента *р на созданный в конце списка элемент *q имеет вид

$p \rightarrow prev = q;$

а последнего на первый

$q \rightarrow next = p;$

РЕАЛИЗАЦИЯ СПИСКА

- Вышеописанная реализация списка в виде связанной динамической структуры имеет ряд очевидных достоинств
- К числу этих достоинств относятся:
 - возможность создавать, удалять и регулировать размер списков во время выполнения программы;
 - относительная простота выполнения операций добавления элементов в список и их удаления из списка

РЕАЛИЗАЦИЯ СПИСКА В МАССИВЕ

- Однако список может быть реализован и с помощью массива
- Для этого необходимо создать массив с типом элемента вида:

```
struct element
```

```
{ int info; // информационное поле
```

```
    int next; // указатель на следующий элемент
```

```
};
```

РЕАЛИЗАЦИЯ СПИСКА В МАССИВЕ

- В этом случае поле ссылки имеет значение индекса следующего элемента
- Для обозначения «свободных» элементов массива можно использовать особые значения поля ссылки, например, равные -2
- Операции добавления новых элементов требуют в этих случаях предварительного поиска в массиве свободных мест

РЕАЛИЗАЦИЯ СПИСКА В МАССИВЕ

- При этом остается ограничение на длину списка, что позволяет реализовать списки с длиной, не превышающей объявленную длину массива
- Соответственно, появляется еще одна дополнительная операция – проверка на *переполнение* списка
- Пример реализации списка в виде массива
- Тестирование приложения

СПИСОК КАК АБСТРАКТНАЯ СТРУКТУРА ДАННЫХ

- Понятие списка вводится в информатике как *структура данных*, представляющая соответствующий *абстрактный тип данных*
- Абстрактным типом данных (АТД) называется тип данных, который определяется путем перечисления набора возможных операций над его данными

АБСТРАКТНЫЕ ТИПЫ ДАННЫХ

- В число этих операций входят операции создания и удаления элементов АД
- Вся внутренняя структура такого типа скрыта от разработчика программного обеспечения и в этом и заключается суть абстракции

СПИСОК КАК АТД

- Конкретные реализации АТД называются структурами данных
- Абстрактный тип данных список может быть реализован при помощи массива или линейного списка
- Однако каждая реализация определяет один и тот же набор функций, который должен работать одинаково (по результату, а не по скорости) для всех реализаций

ПРИМЕР РЕАЛИЗАЦИИ АТД «СПИСОК»

- Текст приложения
- Тестирование приложения