Стеки и очереди.

Стек. Операции со стеком.

Стек – это структура данных (динамическая структура данных, абстрактный тип данных), в которой элементы поддерживают принцип **LIFO («Last in – first out»)**: последним зашёл – первым вышел (или первым зашёл – последним вышел).

Стек позволяет хранить элементы и поддерживает следующие базовые операции:
PUSH – кладёт элемент на вершину стека
POP – снимает элемент с вершины стека, перемещая вершину к следующему элементу
PEEK – получает элемент на вершине стека, но не снимает его оттуда.

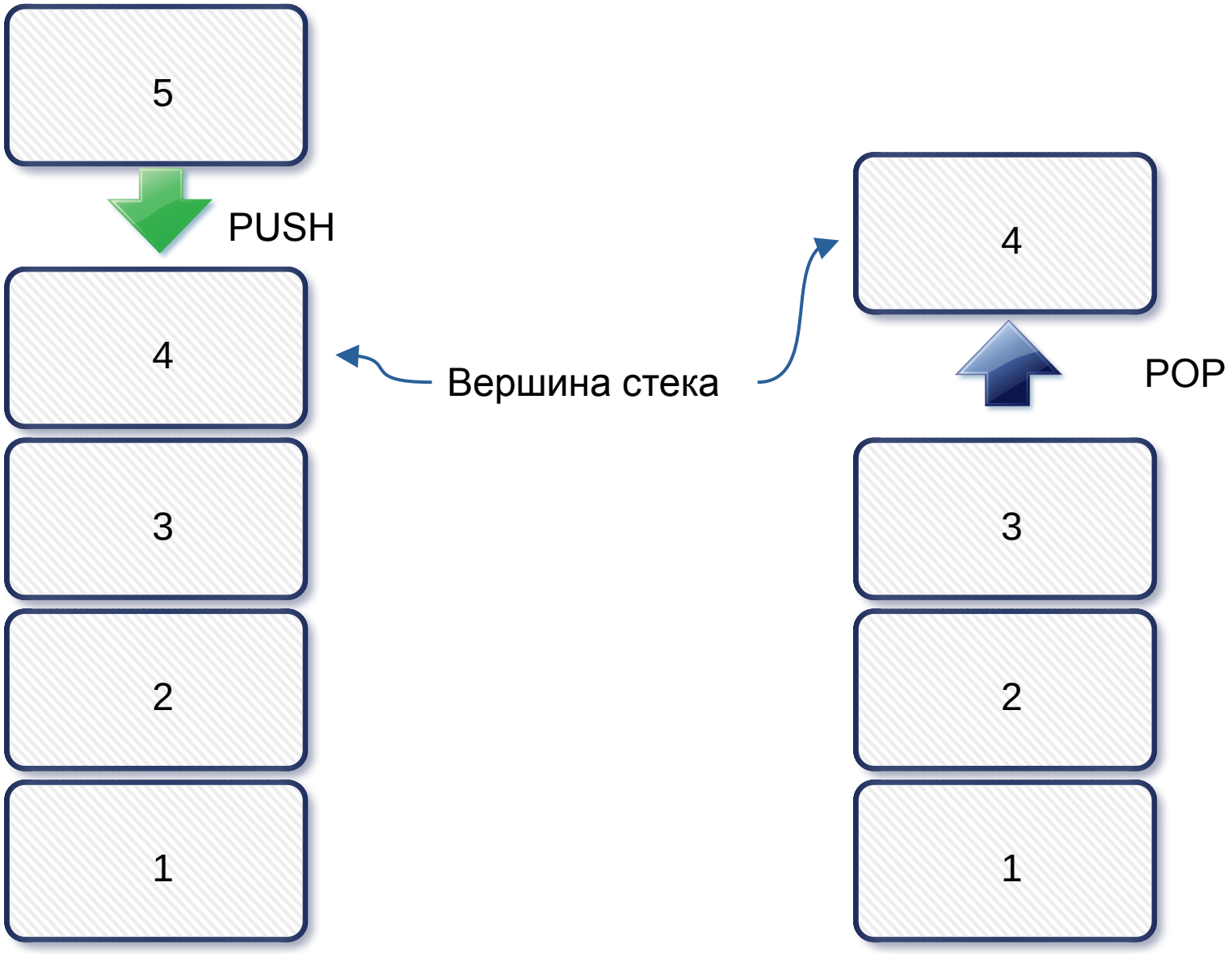
Вершина стека — единственный элемент, с которым можно работать.

Стек. Операции со стеком.

Другими словами:
Стек - это данные динамической структуры, которые представляют собой совокупность линейно-связанных однородных элементов, для которых разрешено добавлять или удалять элементы только с одного конца списка, который называется вершиной (головой) стека.

При записи и выборке изменяется только адрес вершины стека. Поэтому каждый стек имеет базовый адрес, от которого производятся все операции со стеком. В случае, когда стек пуст, адреса вершины и основания стека совпадают.

Стек. Операции со стеком.



Стек. Операции со стеком.

Допустимыми операциями со (над) стеком являются:

- проверка стека на пустоту,
- добавление нового элемента (push)
- изъятие (получение) элемента с «вершины» стека (pop)
- доступ к элементу на «вершине», если стек не пуст (peek).

Стек на односвязном списке

```
struct elem
{
    int id;
    char *data;
    struct elem *next;
};

typedef struct elem item;
```

Если стек строится из элементов типа **item** (как односвязный список), причем имеется «голова» с указателями на первый (**first**) и на последний (**last**) элементы, то можно использовать уже рассмотренные ранее функции работы с односвязным списком (с небольшими модификациями).

push — добавление элемента всегда перед первым
pop — вывод и удаление всегда первого элемента (**head→first**)
peek — вывод первого элемента.

Стек на односвязном списке

Рассмотрим вариант когда стек строится из элементов типа **`item`** (как односвязный список), но у этого списка нет «головы», имеется только текущий адрес вершины — **`top`.**

При добавлении (**`push`**) или удалении (**`pop`**) элементов стека этот адрес меняется внутри соответствующей функции, поэтому передается как параметр по ссылке (через указатель).
А сам элемент стека является указателем на структуру.

Реализация функции push()

```
void push(item **top, char *word, int len, int n)
{
    item *tmp=NULL;
    char *someword=NULL;

    tmp=(item*)malloc(sizeof(item));
    someword=(char*)malloc((len+1)*sizeof(char));

    if(tmp&&someword)
    {
        tmp->next=*top;
        strcpy(someword,word);
        tmp->data=someword;
        tmp->id=n;
        *top=tmp;
    }
}
```

Реализация функции `push()`

Что происходит:

- Выделяем память для элемента стека (для структуры) и для информационных полей, если надо
- Для вновь созданного элемента устанавливаем указатель **`next`** на адрес текущей вершины
- Заполняем информационные поля элемента стека (структуры)
- Указываем адрес текущего элемента как новую вершину стека.

Реализация функции pop()

```
item *pop(item **top)
{
    item *tmp=NULL;

    if(top)
    {
        tmp=*top;
        *top=(*top)->next;
    }
    return tmp;
}
```

Получаем адрес текущей вершины стека, запоминаем его в переменную, делаем вершиной следующий элемент.

При этом количество элементов в стеке уменьшается на 1.

После работы с полученной переменной память, выделенную для нее, нужно очистить обычным способом.

Реализация функции `peek()`

```
item *peek(item *top)
{
    return top;
}
```

Получаем адрес текущей вершины стека, запоминаем его в переменную.

Можно просто посмотреть информационные поля.

Подсчет количества элементов в стеке

Для определения количества элементов в стеке (глубины стека) нужно запомнить адрес вершины в промежуточную переменную-указатель и затем применять переход по **`next`**, пока не получится **`NULL`** («основание» или «дно» стека), на каждом переходе увеличивая счетчик на 1.

Подсчет количества элементов в стеке

```
int elemCount(item *top)
{
    item *tmp;
    int i;
    i=0;
    if(top)
    {
        tmp=top;
        while(tmp)
        {
            tmp=tmp->next;
            i++;
        }
    }
    return i;
}
```

Очистка стека, поиск и удаление элементов

Для очистки стека делаем `pop()` и освобождение памяти для каждого элемента, пока не дойдем до «дна».

См. пример lect-16-01.c

Операции поиска и удаления найденных элементов для стека выполняются точно так же, как и для списка (односвязного).

Очередь. Операции с очередью.

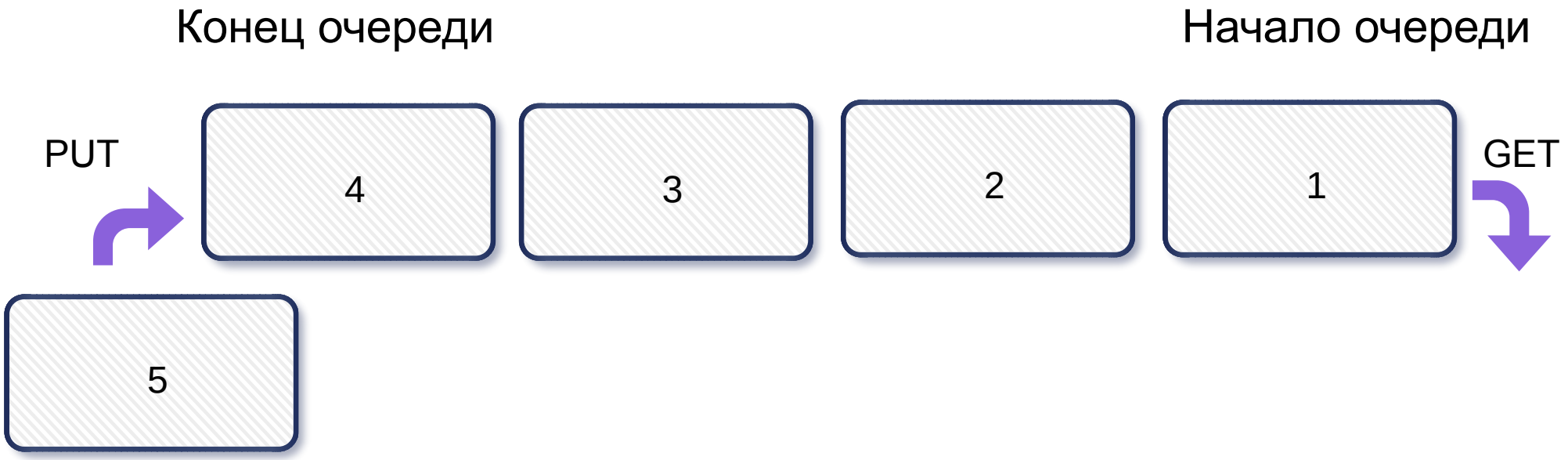
Очередь – структура данных типа «список», позволяющая добавлять элементы лишь в конец списка, и извлекать их из его начала. Она функционирует по принципу **FIFO** (**First In, First Out** — «первым пришёл — первым вышел»).

Элементы добавляются в конец очереди (функция **`put`**), считываются (изымаются) из начала (функция **`get`**).

Типовые операции

- добавление элемента (**`put`**)
- получение (удаление) элемента (**`get`**)
- чтение первого элемента (**`view`**)
- определение длины очереди
- очистка очереди.

Очередь. Операции с очередью.



Очередь на двусвязном списке

```
struct elem
{
    int id;
    char *data;
    struct elem *prev;
    struct elem *next;
};

typedef struct elem item;
```

Если очередь строится из элементов типа **item** (как двусвязный список), причем имеется «голова» с указателями на первый (**first**) и на последний (**last**) элементы, то можно использовать уже рассмотренные ранее функции работы с двусвязным списком (с небольшими модификациями).

put — добавление элемента всегда перед первым
get — вывод и удаление всегда последнего элемента (**head→last**)
view — вывод последнего элемента.

Очередь на двусвязном списке

Рассмотрим вариант когда очередь строится из элементов типа **`item`** (как двусвязный список), но у этого списка нет «головы», имеются только текущие адреса начала (**`qbegin`**) и конца (**`qend`**).

При добавлении (**`put`**) или удалении (**`get`**) элементов очереди эти адреса меняются внутри соответствующих функций, поэтому они передаются как параметры по ссылке (через указатели).
А сами элементы очереди являются указателями на структуры.

Реализация функции `put()`

Что происходит:

- Выделяем память для элемента очереди (для структуры) и для информационных полей, если надо
- Заполняем информационные поля элемента очереди (структуры)
- ***Если очередь пустая*** (в `qbegin` хранится `NULL`)
  - Устанавливаем указатели начала и конца очереди на адрес созданного элемента
  - Для элемента, на который ссылаются начало и конец очереди, устанавливаем `next` и `prev` в `NULL`.
- ***Если очередь не пустая***
  - Делаем новый элемент концом очереди
  - Если в очереди был один элемент, то новый элемент становится предыдущим для элемента в начале очереди.

Реализация функции put()

```
void put(item **qbegin, item **qend, char *word,int len, int n)
{
    item *tmp=NULL;
    char *someword=NULL;

    tmp=(item*)malloc(sizeof(item));
    someword=(char*)malloc((len+1)*sizeof(char));
    if(tmp&&someword)
    {
        strcpy(someword,word);
        tmp->data=someword;
        tmp->id=n;
        if(!(*qbegin)) /* Queue is empty */
        {
            *qbegin=tmp;
            (*qbegin)->next=NULL;
            (*qbegin)->prev=NULL;
            *qend=tmp;
            (*qend)->next=NULL;
            (*qend)->prev=NULL;
        }
        /* else - на следующем слайде       */
```

Реализация функции put()

```
/* void put - окончание */

        else
        {
            tmp->next=*qend;
            (*qend)->prev=tmp;
            *qend=tmp;
            if(!((*qbegin)->prev)) (*qbegin)->prev=tmp;
        }
    }
}
```

Реализация функции `get()`

Получаем адрес текущего начала очереди, запоминаем его в переменную, делаем началом очереди предыдущий элемент.
Если начало очереди не `NULL` (еще есть элементы в очереди), то указываем, что элемент в начале очереди — последний.

Если элементов в очереди больше нет, то адрес начала очереди тоже нужно установить в `NULL`.

При выполнении функции `get()` количество элементов в очереди уменьшается на 1.

После работы с полученной переменной память, выделенную для нее, нужно очистить обычным способом.

Реализация функции get()

```
item *get(item **qbegin, item **qend)
{
    item *tmp=NULL;

    if(*qbegin)
    {
        tmp=*qbegin;
        *qbegin=(*qbegin)->prev;
        if(*qbegin)
        {
            if((*qbegin)->next) (*qbegin)->next=NULL;
        }
        else *qend=NULL;
    }
    return tmp;
}
```

Реализация функции `view()`

```
item *view(item *qbegin)
{
    return qbegin;
}
```

Получаем адрес элемента в начале очереди, запоминаем его в переменную.

Можно просто посмотреть информационные поля.

Подсчет количества элементов в очереди

Для определения количества элементов в очереди (длины очереди) нужно запомнить адрес конца очереди в промежуточную переменную-указатель и затем применять переход по **`next`**, пока не получится **`NULL`** (признак начала очереди), на каждом переходе увеличивая счетчик на 1.

Подсчет количества элементов в очереди

```
int elemCount(item *qend)
{
    item *tmp;
    int i;
    i=0;
    if(qend)
    {
        tmp=qend;
        while(tmp)
        {
            tmp=tmp->next;
            i++;
        }
    }
    return i;
}
```

Очистка очереди, поиск и удаление элементов

Для очистки очереди делаем **`get()`** и освобождение памяти для каждого элемента, пока очередь не станет пустой.

См. пример lect-16-02.c

Операции поиска и удаления найденных элементов для очереди выполняются точно так же, как и для списка (двусвязного).

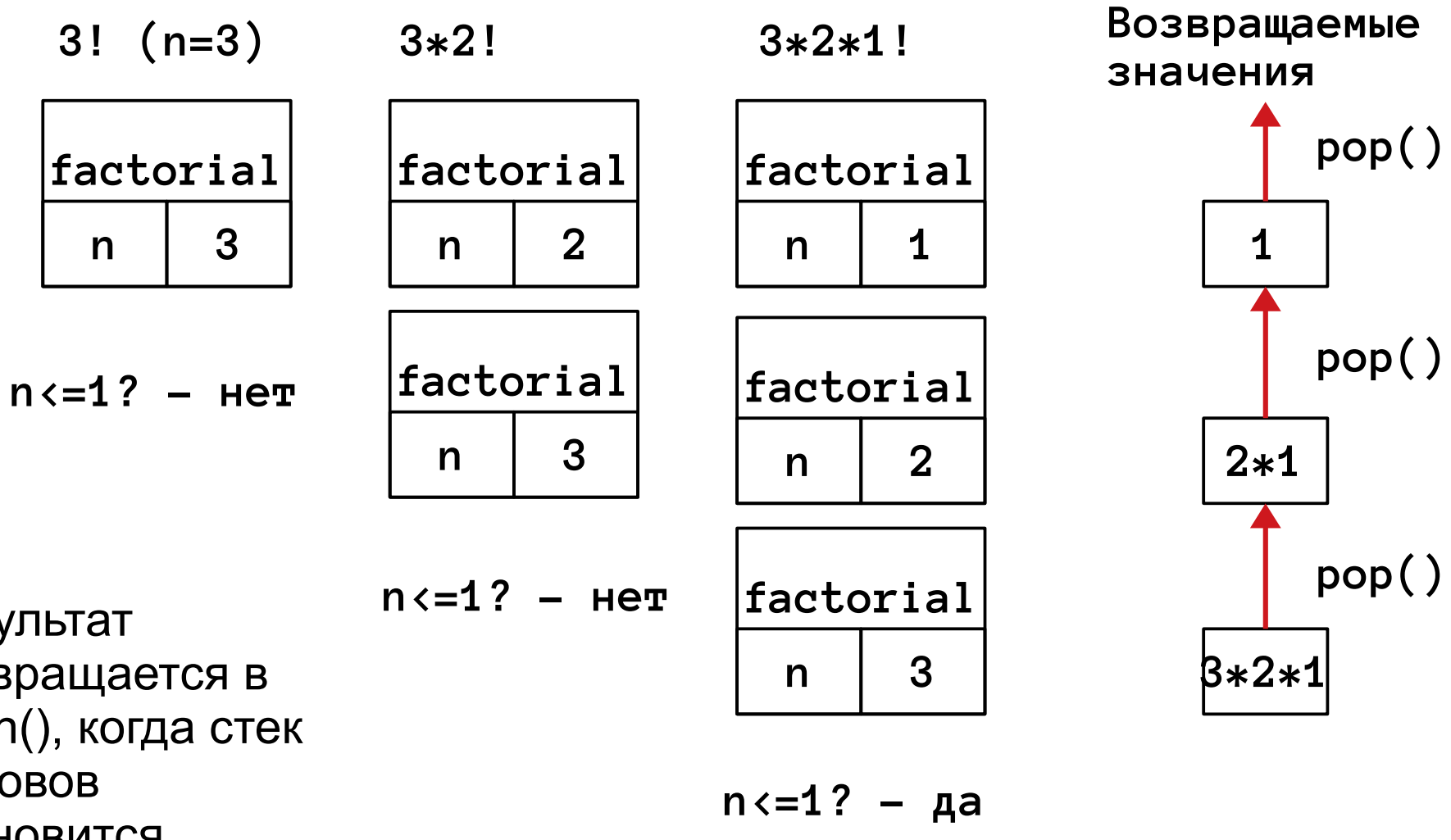
Стек вызовов для рекурсивных функций

Пример рекурсивного алгоритма — вычисление факториала

Определение (мат. формула): **N!=N*(N-1)!**

```
long factorial(int n)
{
    long f;
    if (n<0) f=0; /* to prevent input errors */
    else if(n<=1) f=1; /* condition to stop calculations */
    else f=n*factorial(n-1); /* recursive call */
    return f;
}
```

Стек вызовов



Результат возвращается в main(), когда стек вызовов становится пустым.

Опасность: переполнение стека памяти (все функции — в стеке!)

Рекурсия на списке

Пусть имеется односвязный линейный список, структура элементов и типы была рассмотрены ранее.

```
struct elem
{
    int id;
    char *data;
    struct elem *next;
};

typedef struct elem item;
```

Сделаем функции заполнения списка и вывода списка рекурсивными.

Рекурсия на списке

Для рекурсивного заполнения (функция **`put()`**):

Граничный случай — ввод пустой строки как поля данных текущего элемента. Тогда возвращается указатель **`NULL`** (конец списка).

Рекурсивный случай — каждый новый элемент становится следующим элементом списка (добавление после последнего). Возвращается указатель на первый элемент.

Значения для заполнения поля **`data`** для структуры в элементах списка вводятся пользователем, поле **`id`** формируется автоматически (автоинкремент).

Рекурсия на списке

Функция put()

```
item *put(int n)
{
    enum {maxlen=64};
    item *tmp=NULL;
    char *someword=NULL;
    char myword[maxlen];
    int len;

    printf("Your word: ");
    fgets(myword,maxlen,stdin);
    len=strlen(myword);
    myword[len-1]='\0';
    n++;
    tmp=(item*)malloc(sizeof(item));
    someword=(char*)malloc((len+1)*sizeof(char));

 /* продолжение – следующий слайд */
```

Рекурсия на списке

Функция `put() /* продолжение */`

```
    if((len-1)==0)
    {
        free(tmp);
        tmp=NULL;
    }
    else
    {
        if(tmp&&someword)
        {
            strcpy(someword,myword);
            tmp->data=someword;
            tmp->id=n;
            tmp->next=put(n);
        }
    }
    return tmp;
}
```

Рекурсия на списке

Для рекурсивного получения элементов (функция **`get()`**):

Граничный случай — получен указатель с значением NULL (список кончился).

Рекурсивный случай — выводятся информационные поля текущего элемента, делается переход в следующему элементу, текущий элемент очищается.

Рекурсия на списке

Функция `get()`

```
void get(item *qb)
{

    if(qb==NULL) printf("End of queue!\n");
    else
    {
        printf("id: %d \tword: %s\n",qb->id,qb->data);
        get(qb->next);
        free(qb);
        qb=NULL;
    }
}
```

Рекурсия на списке

Эти две функции (**put()** и **get()**) реализуют очередь на односвязном списке (дисциплина FIFO) – пример lect-16-03.c.

```
int main()
{
    item *q=NULL;
    int n;
    n=0;

    q=put(n);
    puts("\nEntering finished");
    n=elemCount(q); // была определена раньше
    printf("Elements in queue: %d\n",n);
    puts("\nGet all elements:");
    get(q);
    return 0;
}
```