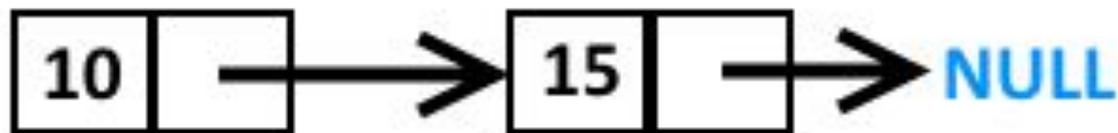


# Сложные структуры данных

## Связные списки

# Структуры, ссылающиеся на

```
struct node {  
    int x;  
    struct node *next;  
};
```



# СВЯЗНЫЙ СПИСОК

- Структура данных, представляющая собой конечное множество упорядоченных элементов (узлов), связанных друг с другом посредством указателей, называется СВЯЗНЫМ СПИСОКОМ.
- Каждый элемент связного списка содержит поле с данными, а также указатель на следующий и/или предыдущий элемент. Эта структура позволяет эффективно выполнять операции добавления и удаления элементов для любой позиции в последовательности.

# Недостатки связного списка

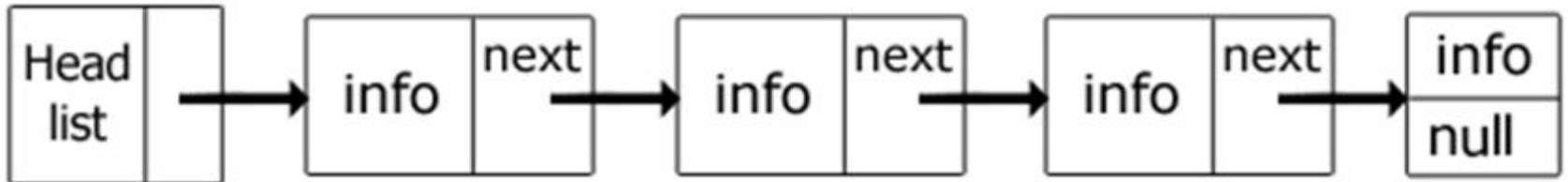
- Недостатком связного списка, как и других структур типа «список», в сравнении его с массивом, является отсутствие возможности работать с данными в режиме произвольного доступа, т. е. список – структура последовательно доступа, в то время как массив – произвольного.

# Односвязный список

- Каждый узел односвязного (однонаправленного связного) списка содержит указатель на следующий узел. Из одной точки можно попасть лишь в следующую точку, двигаясь тем самым в конец. Так получается своеобразный поток, текущий в одном направлении.

# Односвязный список

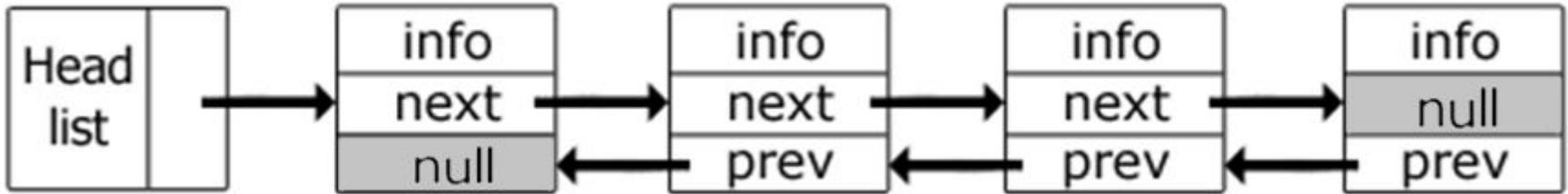
- Каждый из блоков представляет элемент (узел) списка. Здесь и далее Head list – заголовочный элемент списка (для него предполагается поле next). Он не содержит данные, а только ссылку на следующий элемент. На наличие данных указывает поле info, а ссылки – поле next. Признаком отсутствия указателя является поле null



# Односвязные и двусвязные СПИСКИ

- Односвязный список не самый удобный тип связного списка, т. к. из одной точки можно попасть лишь в следующую точку, двигаясь тем самым в конец. Когда в двусвязном списке, кроме указателя на следующий элемент есть указатель на предыдущий.

# Двусвязный список



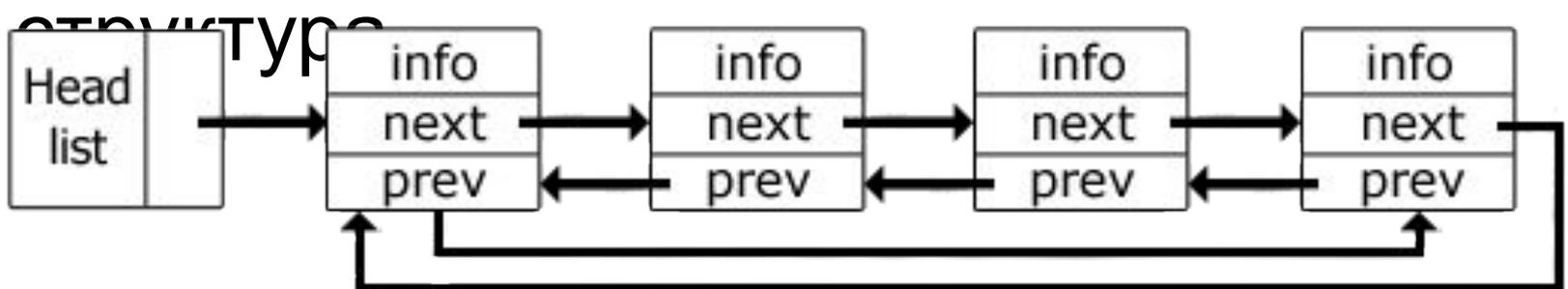
- Особенность двусвязного списка, что каждый элемент имеет две ссылки: на следующий и на предыдущий элемент, позволяет двигаться как в его конец, так и в начало.
- Операции добавления и удаления здесь наиболее эффективны, чем в односвязном списке, поскольку всегда известны адреса тех элементов списка, указатели которых направлены на изменяемый элемент. Но добавление и удаление элемента в двусвязном списке, требует изменения большого количества ссылок, чем этого потребовал бы односвязный список.

# Двусвязный список

- Возможность двигаться как вперед, так и назад полезна для выполнения некоторых операций, но дополнительные указатели требуют задействования большего количества памяти, чем таковой необходимо в односвязном списке.

# Кольцевой список

- Еще один вид связанного списка – **кольцевой** список. В кольцевом односвязном списке последний элемент ссылается на первый. В случае двусвязного кольцевого списка – плюс к этому первый ссылается на последний. Таким образом, получается зацикленная



# Управление памятью

- Для создания и использования динамических структур требуется ***динамическое распределение памяти*** — возможность получения памяти для хранения новых узлов и освобождать память для удаления узлов.

# Управление памятью

- Функции для управления динамической памятью объявлены в **stdlib.h**
- функция для выделения памяти:  
`void* malloc (size_t size);`
- Функция для освобождения ранее выделенной памяти:  
`void free (void* ptr);`

# malloc

**void\* malloc (size\_t size);**

Резервирует блоки памяти размером `size` байт памяти и возвращает указатель на начало зарезервированного участка памяти.

Например:

```
newPtr = malloc (sizeof(struct node));
```

`sizeof(struct node)` определяет размер в байтах структуры типа `struct node`, а `malloc` выделяет новый блок памяти размером в `sizeof(struct node)` и возвращает указатель на выделенную память в `newPtr`. Если памяти для выделения не достаточно, то `malloc` возвращает указатель на `NULL`

# free

```
void free (void* ptr);
```

Освобождает память, т.е. память возвращается системе, и в дальнейшем её можно будет выделить снова.

Например:

```
free (newPtr);
```

После того как выделенная память больше не нужна необходимо её освободить при помощи free. Так же это необходимо делать перед завершением программы, если память ещё не была освобождена.

```
#include <stdlib.h>
```

```
struct node {
```

```
    int x;
```

```
    struct node *next;
```

```
};
```

```
int main()
```

```
{
```

```
    /* Обычная структура */
```

```
    struct node *root;
```

```
    /* Теперь root указывает на структуру node */
```

```
    root = (struct node *) malloc( sizeof(struct node) );
```

```
    /* Узел root указывает на следующий элемент, которого пока  
нет */
```

```
    root->next = NULL;
```

```
    /* Использование оператора -> позволяет изменять узел  
структуры, на которую он указывает */
```

```
    root->x = 5;
```

```
    free ( root );
```

```
    return 0;
```

```
}
```

```
int main()
{
    /* Это менять нельзя, т.к. тогда мы потеряем список в памяти
    */
    struct node *root;
    /* Это указывает на каждый элемент списка, пока мы
    пробегаем по нему */
    struct node *conductor;

    root = malloc( sizeof(struct node) );
    root->next = NULL;
    root->x = 12;
    conductor = root;
    if ( conductor != NULL ) {
        while ( conductor->next != NULL )
        {
            conductor = conductor->next;
        }
    }
}
```

```
/* Создаёт новый узел в конце */
conductor->next = malloc( sizeof(struct node) );

conductor = conductor->next;

if ( conductor == NULL )
{
    printf("Не хватает памяти!\n");
    return 0;
}
/* инициализируем память */
conductor->next = NULL;
conductor->x = 42;

return 0;
}
```

```
conductor = root;
if ( conductor != NULL ) {
/*убедимся, что существует место старта*/
    while ( conductor->next != NULL ) {
        printf( "%d\n", conductor->x);
        conductor = conductor->next;
    }
    printf( "%d\n", conductor->x );
}
```

```
conductor = root;  
while ( conductor != NULL ) {  
    printf( "%d\n", conductor->x );  
    conductor = conductor->next;  
}
```

# Очистка памяти

```
struct node *temp;  
temp = root->ptr;  
free(root); /* освобождение памяти  
текущего корня*/  
root = temp; // новый корень списка
```

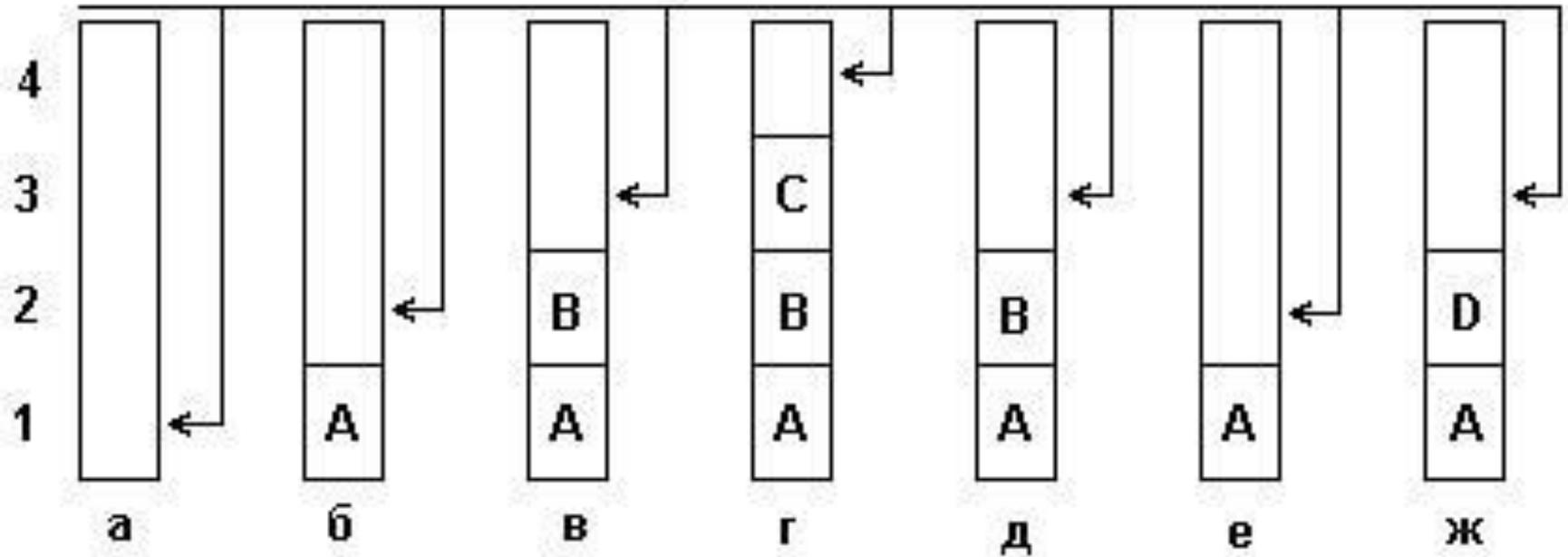
# Стек

*Стеком* называется структура данных, организованная по принципу ***LIFO – last-in, first-out***, т.е. элемент, попавшим в множество последним, должен первым его покинуть.

При практическом использовании часто налагается ограничение на длину стека, т.е. требуется, чтобы количество элементов не превосходило заранее определенное целое число ***N***

# Стек

Вершина стека



# Стек

## Реализация 1

на основе массива

Для реализации стека, состоящего не более чем из **100** чисел, следует определить массив, состоящий из **100** элементов и целую переменную, указывающую на вершину стека (ее значение будет также равно текущему числу элементов в стеке)

```
int stack[100], n=0;
```

# Стек

## Реализация 2

на основе массива с использованием  
общей структуры

```
struct Stack{  
    int stack[100];  
    int n;  
};
```

# Стек

## Реализация 3

на основе указателей

Если максимальный размер стека можно выяснить только после компиляции программы, то память под стек нужно выделять **динамически**. При этом, вершину стека можно указывать не индексом в массиве, а указателем на вершину. Для этого следует определить следующие переменные

```
int *stack, *head;
```

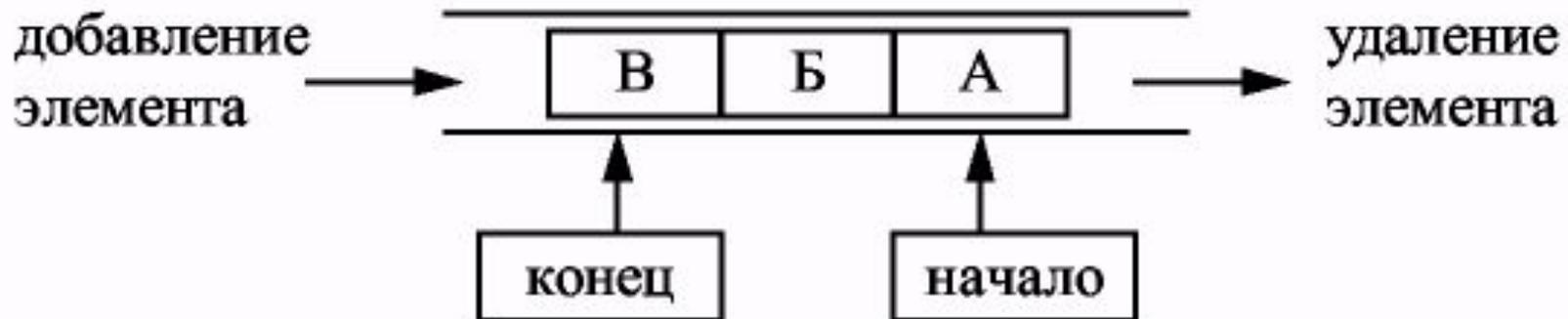
или

```
struct SStack{  
    int *stack;  
    int *n;  
};
```

# Очередь

Очередью называется структура данных, организованная по принципу *FIFO – first-in, first-out*, т.е. элемент, попавшим в множество первым, должен первым его и покинуть. При практическом использовании часто налагается ограничение на длину очереди, т.е. требуется, чтобы количество элементов не превосходило заранее определенное целое число  $N$

# Очередь



# Очередь реализация

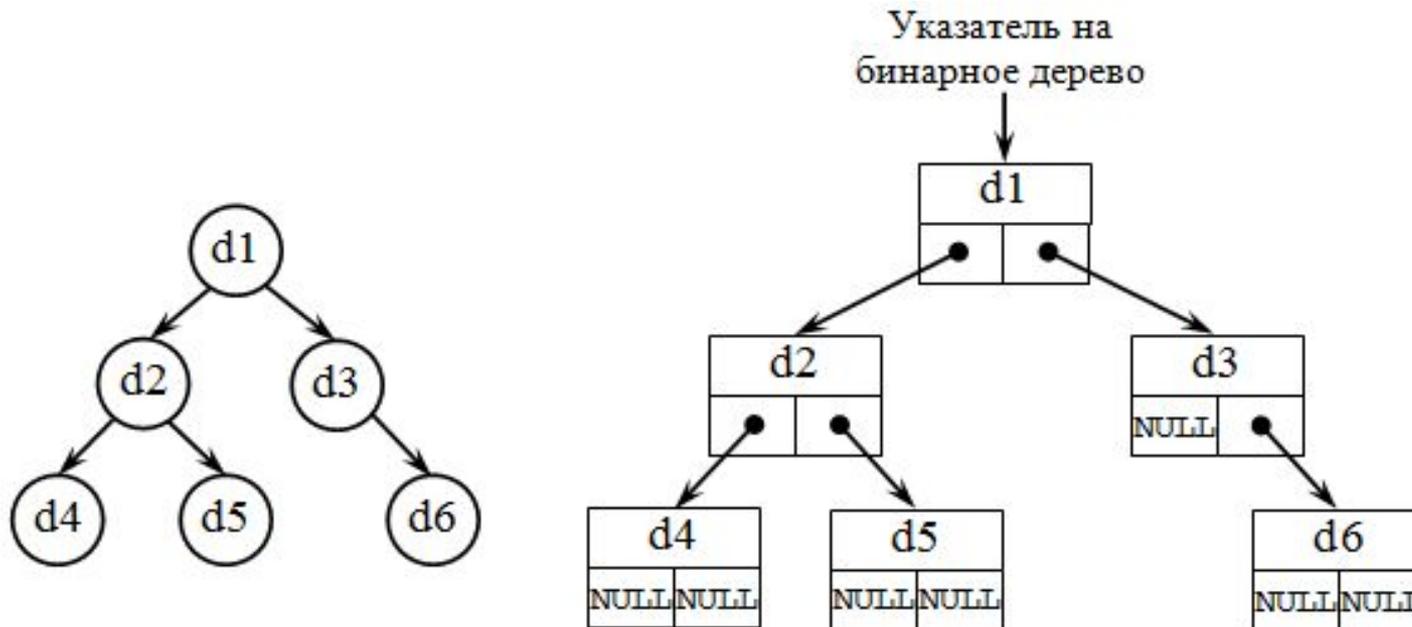
Для реализации очереди необходимо знать первый и последний элемент находящийся в ней. Например, для реализации стандартной очереди из менее чем 100 целых чисел определить следующие данные

```
#define N 100  
int array[N], begin=0, end=0;
```

Соответственно при добавлении элементов в очередь переменная **end** будет увеличиваться, а при изъятии их из очереди увеличиваться будет **begin**.

# Бинарное дерево

***Бинарными деревьями*** называют структуру данных, в которой, как правило, задан *корневой элемент* или *корень* и для каждой вершины (элемента) существует не более двух *потомков*.



# Бинарное дерево реализация

```
struct STree{  
    int value;  
    struct STree *prev;  
    struct STree *left, *right;  
};
```

здесь указатель *prev* указывает на родительский элемент данной вершины, а *left* и *right* – на двух потомков, которых традиционно называют **левым** и **правым**. Величина *value* называется **ключом** вершины.

# Бинарное дерево

Бинарное дерево называется **деревом поиска**, если для любой вершины дерева  $a$  ключи всех вершин в правом поддереве больше или равны ключа  $a$ , а в левом – меньше. Неравенства можно заменить на строгие, если известно, что в дереве нет равных элементов.

# Дерево поиска

## Поиск элемента

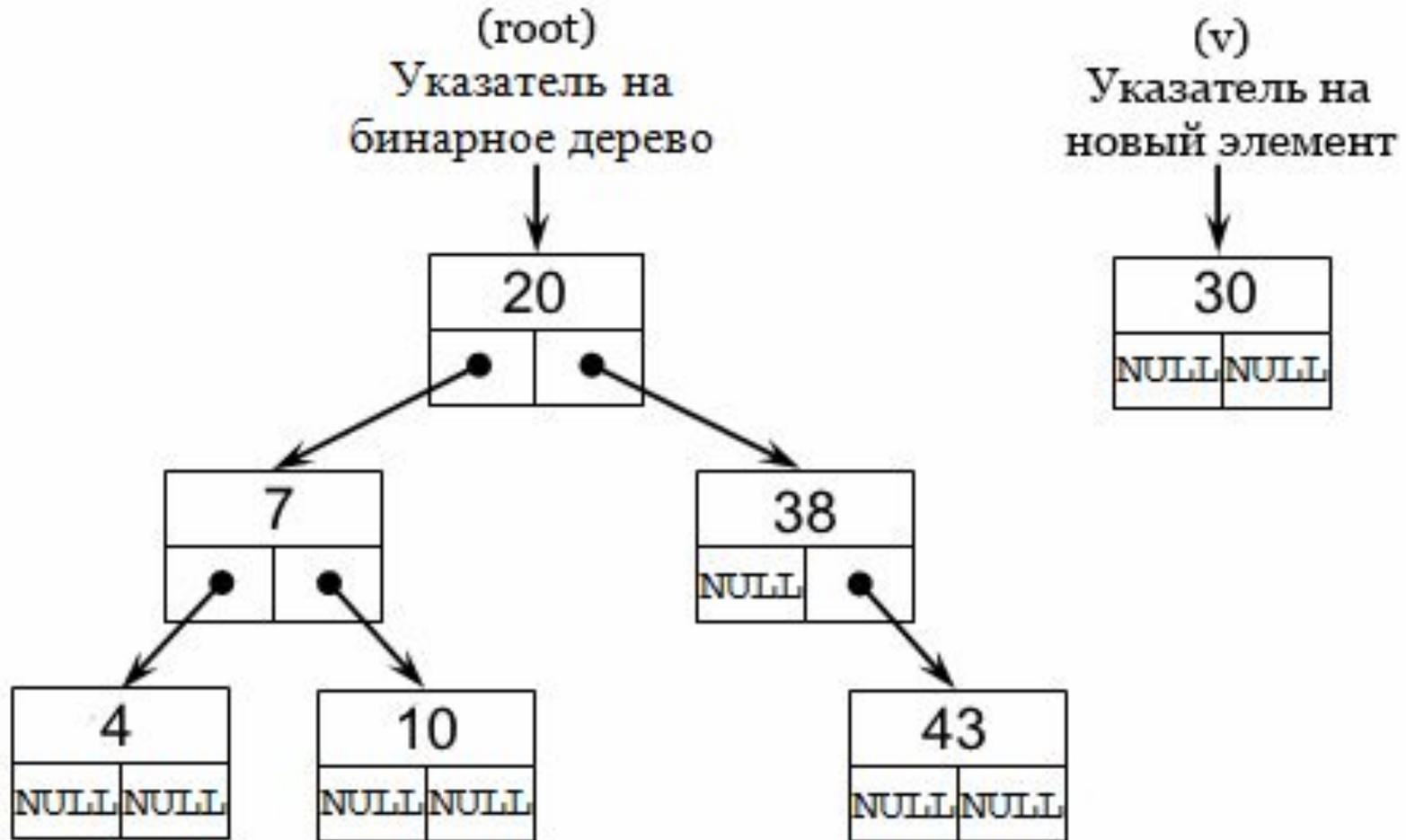
```
struct STree *Find(struct STree *root, int v){  
    if(root==NULL)return NULL;  
    if(root->value==v)return root;  
    if(root->value>=v)return Find(root->right,v);  
    else return Find(root->left, v);  
};
```

# Дерево поиска

## Добавление элемента

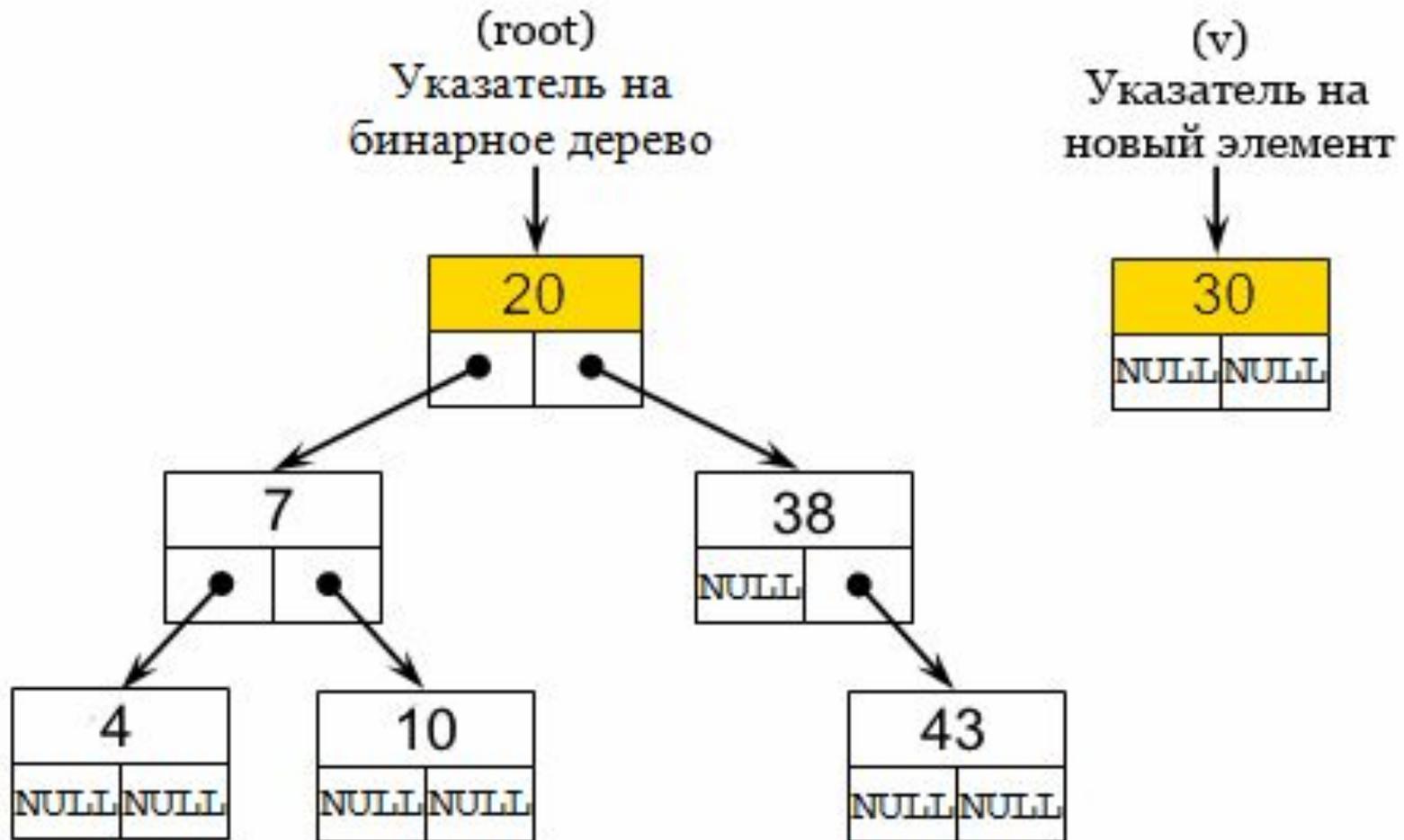
```
struct STree *Insert(struct STree *root, struct STree *v){
    if(v->value>=root->value)
        return root->right==NULL ?
            (v->prev=root, v->right=v->left=NULL, root->right=v) :
            Insert(root->right, v);
    else
        return root->left==NULL ?
            (v->prev=root, v->right=v->left=NULL, root->left=v) :
            Insert(root->left, v);
};
```

# Дерево поиска



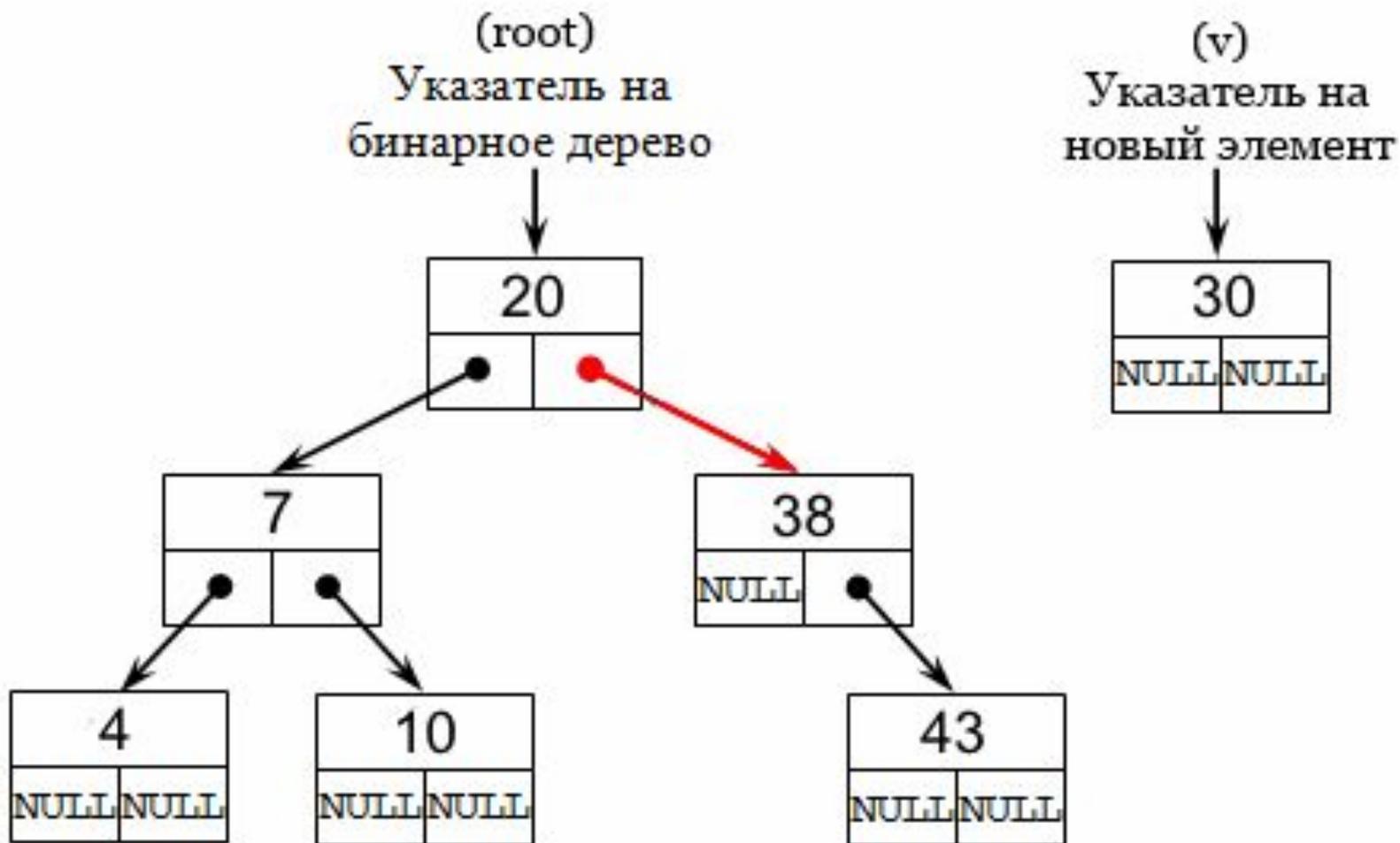
# Дерево поиска

`if(v->value >= root->value)`



# Дерево поиска

**return** root->right==**NULL** ?



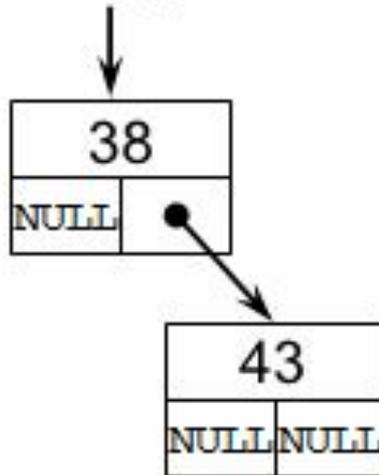
# Дерево поиска

root->right==**NULL** ?

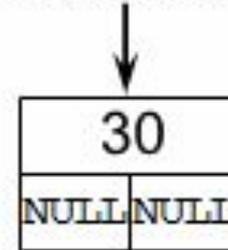
(v->back=root, v->right=v->left=**NULL**, root->right=v) :

Insert(root->right, v);

(root->right)  
указатель на  
ветвь дерева



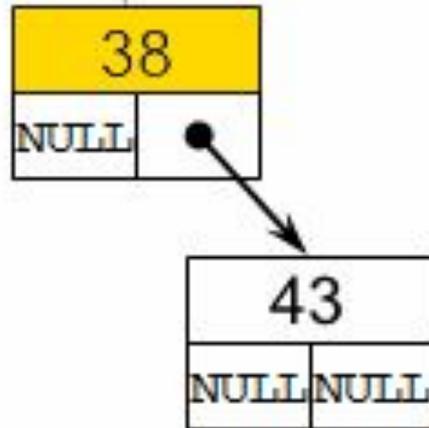
(v)  
Указатель на  
новый элемент



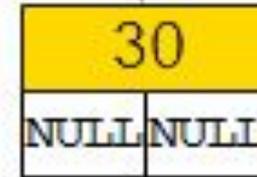
# Дерево поиска

`if(v->value >= root->value)`

(root->right)  
указатель на  
ветвь дерева



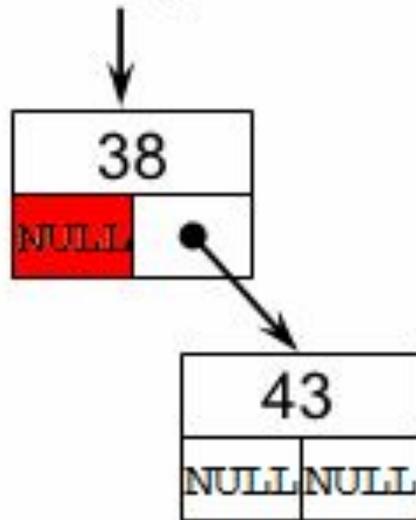
(v)  
Указатель на  
НОВЫЙ ЭЛЕМЕНТ



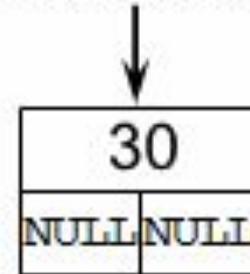
# Дерево поиска

**return** root->left==**NULL** ?

(root->right)  
указатель на  
ветвь дерева



(v)  
Указатель на  
новый элемент

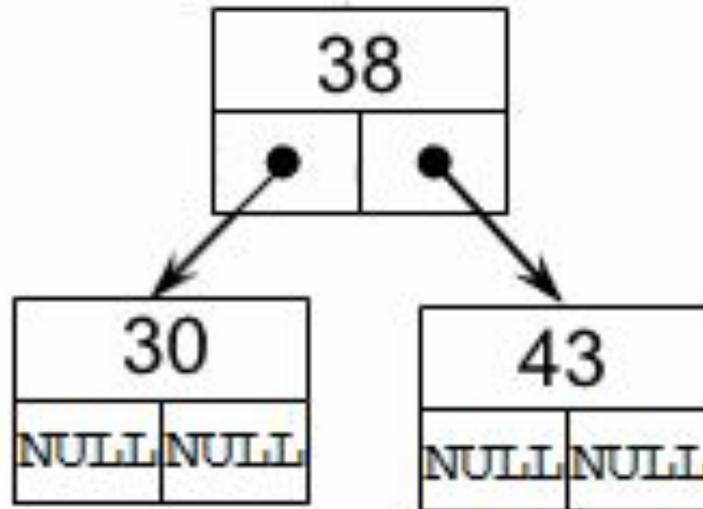


# Дерево поиска

root->left==**NULL** ?

(v->prev=root, v->right=v->left=**NULL**, root->left=v) :

Insert(root->left, v);



# Дерево поиска

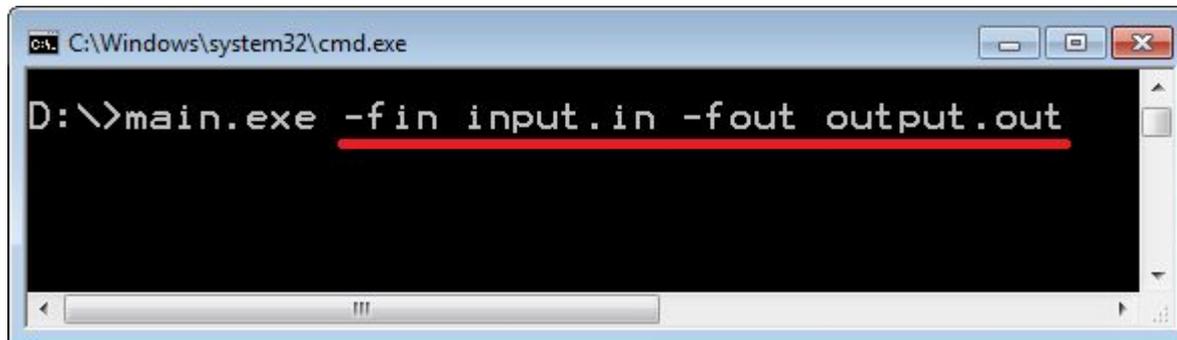
## Добавление элемента

Функция **Insert** добавляет элемент в бинарное дерево поиска и возвращает указатель на добавляемый элемент.

# Аргументы командной строки

При запуске программы через консоль возможно передать в программу данные, называемые **Аргументы командной строки**, в виде строк.

Они могут быть использованы во время работы программы



```
C:\Windows\system32\cmd.exe
D:\>main.exe -fin input.in -fout output.out
```

The image shows a screenshot of a Windows command prompt window. The title bar reads 'C:\Windows\system32\cmd.exe'. The command prompt shows the current directory as 'D:\>' and the command 'main.exe -fin input.in -fout output.out' has been entered. A red underline is drawn under the arguments '-fin input.in -fout output.out'.

# Аргументы командной строки

Обратиться к аргументам командной строки в программе возможно через специальные переменные `int argc` и `char *argv[]`

`argc` – число переданных аргументов,  
`argv` – массив строк равный числу аргументов.

При вызове программы всегда есть один аргумент имя запущенной программы.

# Аргументы командной строки

```
#include <stdio.h>
```

```
int main(int argc, char *argv){  
    if(argc > 1) {  
        printf("Вы ввели много чего");  
    }  
    printf("Но это точно имя этой программы: %s", argv[0]);  
  
    return 0;  
}
```

# Аргументы командной строки

