

# Структуры данных в C++

---

Бирюков В. А.

November 5, 2022

- Статический массив
- Динамический массив
- Связный список
- Дерево
- Хеш-таблица

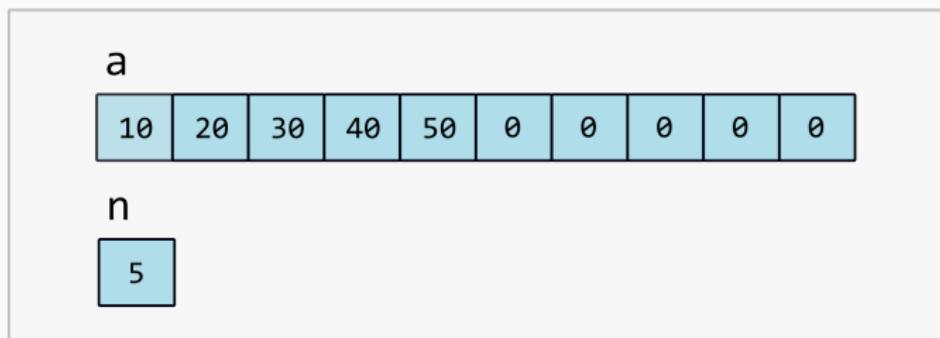
## **Статический массив**

---

# Статический массив

```
int a[10] = {10, 20, 30, 40, 50};  
int n = 5;
```

## Стек



# Статический массив

```
std::array<int, 10> a {10, 20, 30, 40, 50};  
int n = 5;
```

## Стек

a

10	20	30	40	50	0	0	0	0	0
----	----	----	----	----	---	---	---	---	---

n

5

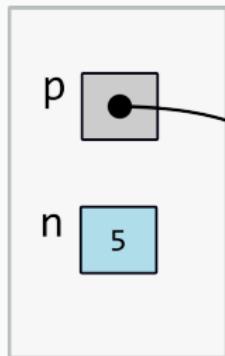
## Динамический массив

---

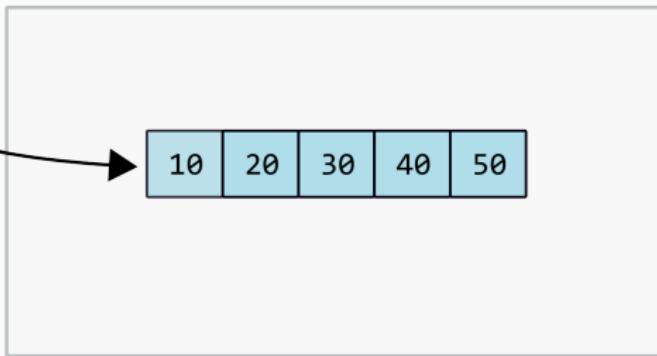
# Динамический массив в языке С

```
#include <stdlib.h>  
...  
int n = 5;  
int* p = (int*)malloc(n * sizeof(int));  
for (int i = 0; i < n; ++i)  
    p[i] = (i + 1) * 10;
```

Стек

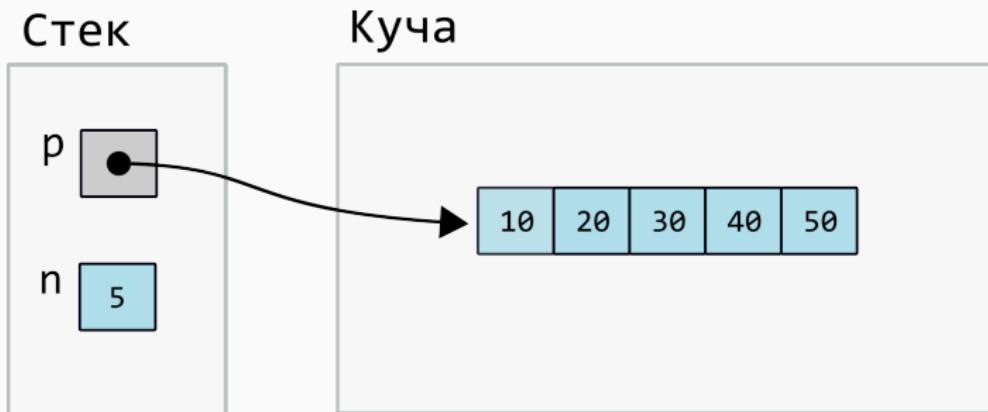


Куча



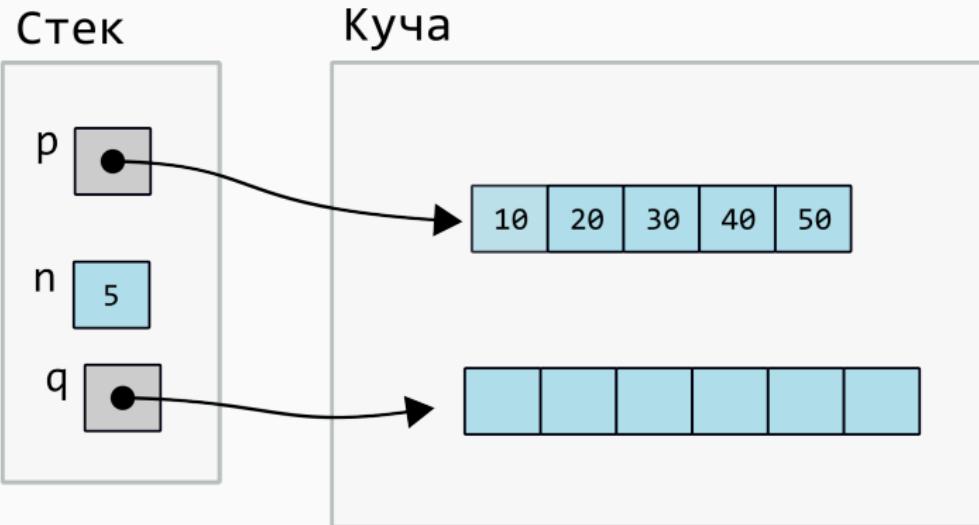
# Динамический массив в языке C++ с помощью new

```
int* p = new int[5] {10, 20, 30, 40, 50};  
int n = 5;
```



## Добавим ещё один элемент в динамический массив

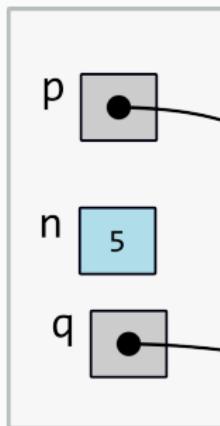
```
int* p = new int[5] {10, 20, 30, 40, 50};  
int n = 5;  
int* q = new int[n + 1];
```



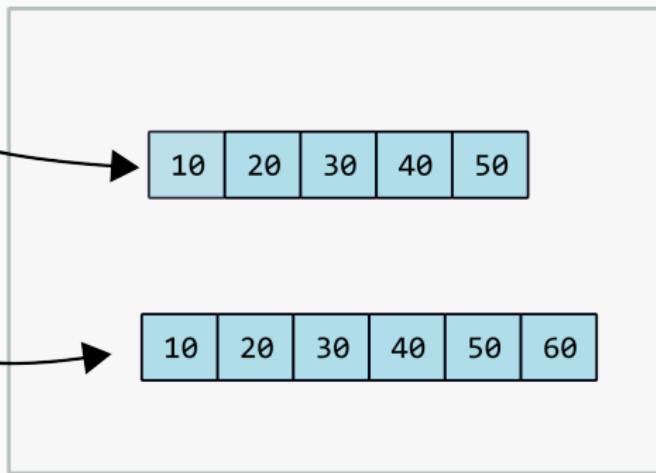
# Добавим ещё один элемент в динамический массив

```
for (int i = 0; i < n; ++i)
    q[i] = p[i];
q[n] = 60;
```

Стек



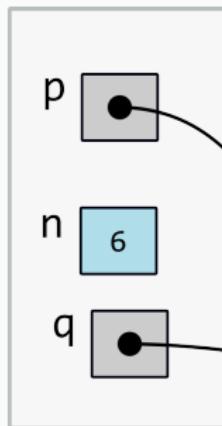
Куча



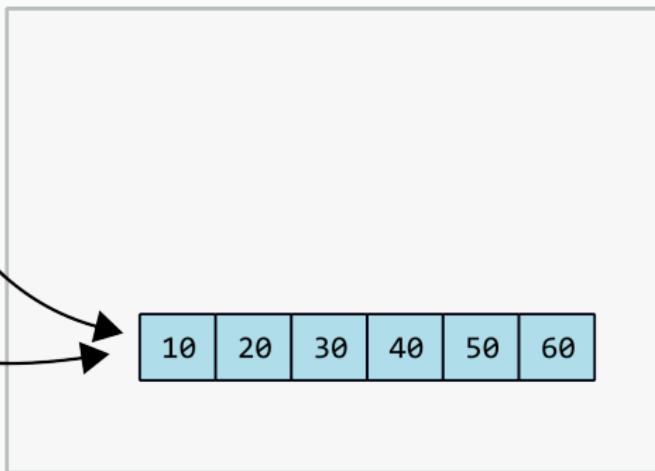
# Добавим ещё один элемент в динамический массив

```
delete p;  
p = q;  
n += 1;
```

Стек



Куча

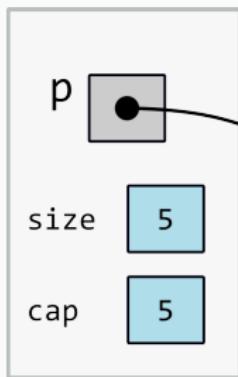


## Добавим ещё один элемент в динамический массив

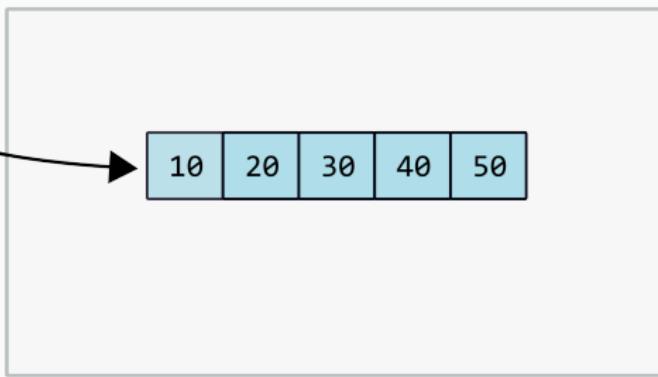
- Предположим, что мы захотели добавить в динамический массив ещё 100 элементов. Тогда при каждом добавлении элемента мы будем должны выделять в куче массив на один элемент больше и переписывать весь массив в новое место. Вычислительная сложность добавления  $N$  элементов в такой массив будет равна  $O(N^2)$ .
- Предположим, что мы захотели удалить 1 элемент из конца массива. Тогда нам тоже придётся перевыделять память, хотя без этого можно обойтись.

```
int* p = new int[5] {10, 20, 30, 40, 50};  
int size = 5;  
int cap = 5;
```

Стек



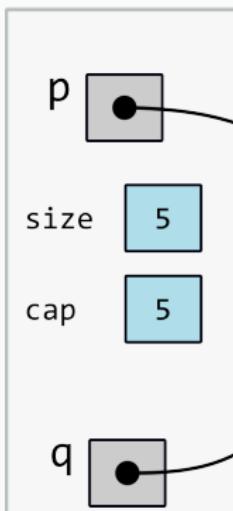
Куча



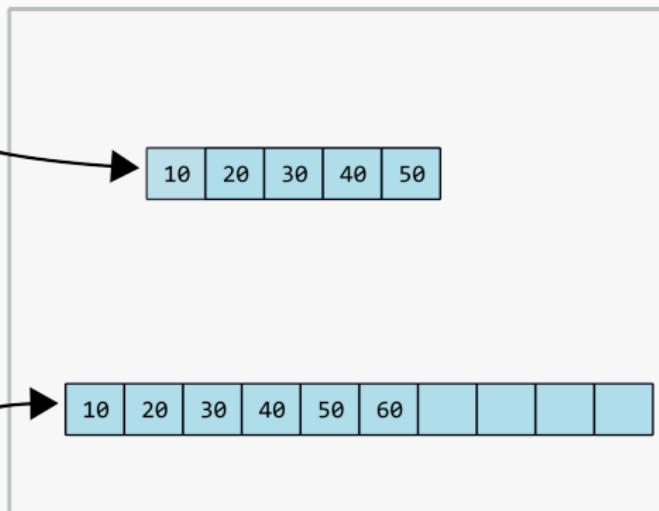
# Добавление элемента

```
int* q = new int[2 * n];
for (int i = 0; i < size; ++i)
    q[i] = p[i];
q[size]++;
```

Стек



Куча



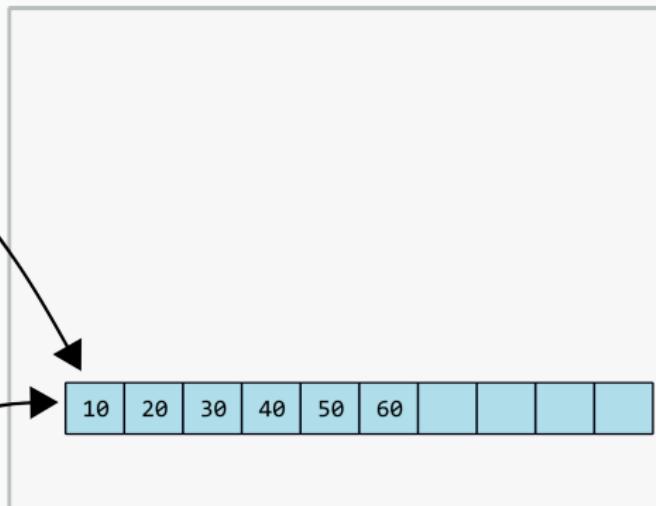
# Добавление элемента

```
delete p;  
p = q;  
size += 1;  
cap *= 2;
```

Стек

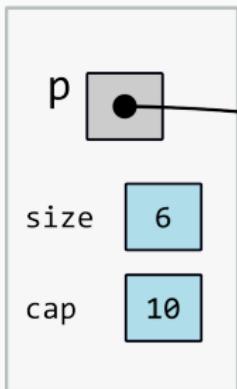


Куча

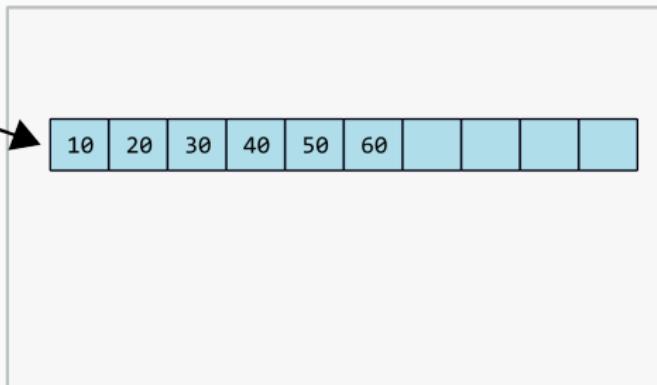


# Добавление элемента

Стек

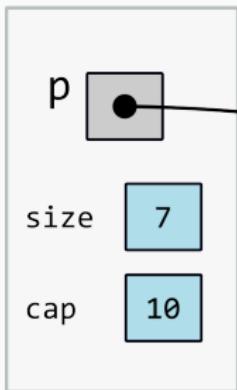


Куча

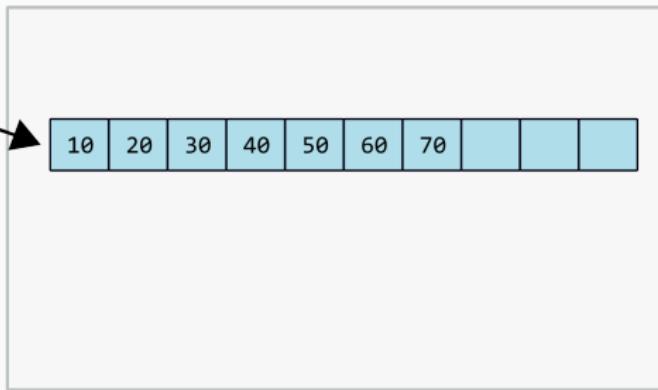


# Добавление ещё одного элемента

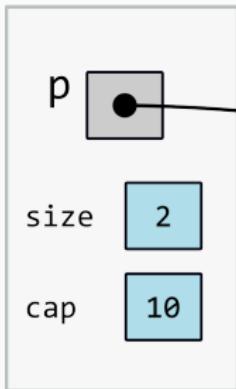
Стек



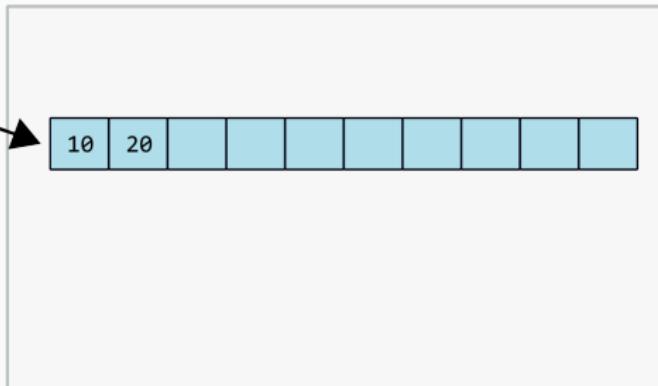
Куча



Стек



Куча

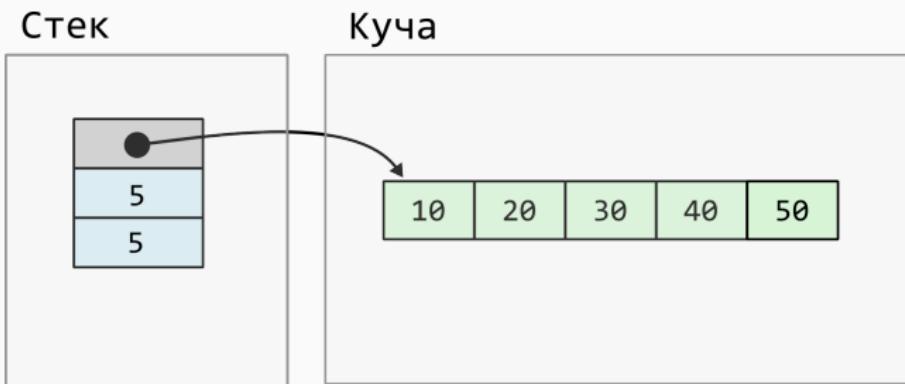


## Динамический массив

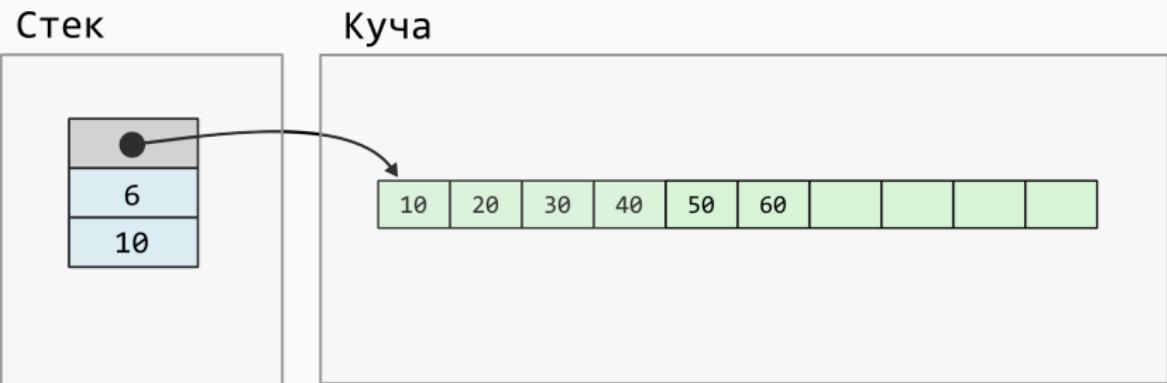
`std::vector`

---

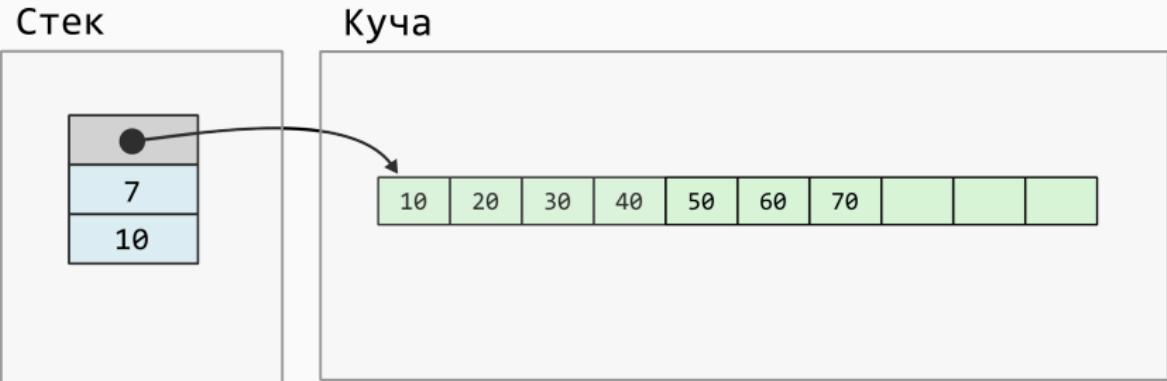
```
std::vector<int> v {10, 20, 30, 40, 50};
```



```
std::vector<int> v {10, 20, 30, 40, 50};  
v.push_back(60);
```



```
v.push_back(70);
```



## **Связный список**

---

# Связный список



```
struct Node
{
    int value;
    Node* next;
};
```

# Связный список



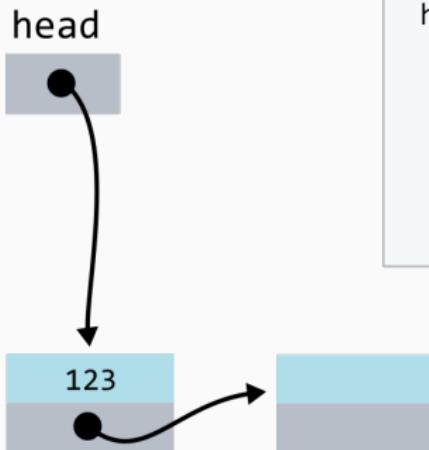
```
Node* head = malloc(sizeof(Node));
```

# Связный список



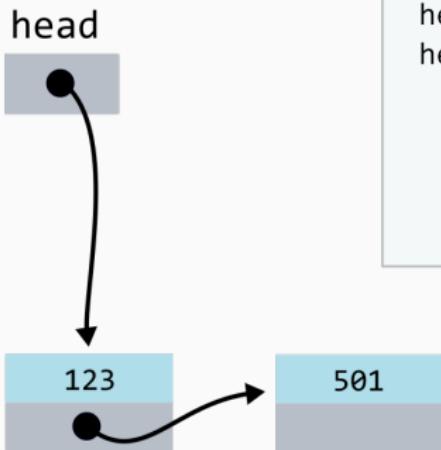
```
Node* head = malloc(sizeof(Node));  
head->value = 123;
```

# Связный список



```
Node* head = malloc(sizeof(Node));  
head->value = 123;  
head->next = malloc(sizeof(Node));
```

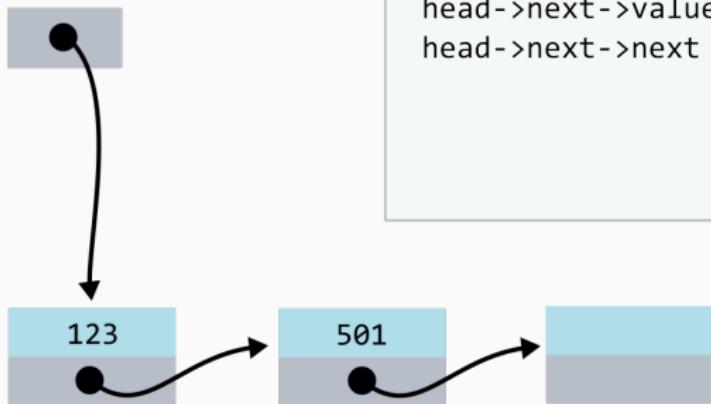
# Связный список



```
Node* head = malloc(sizeof(Node));  
head->value = 123;  
head->next = malloc(sizeof(Node));  
head->next->value = 501;
```

# Связный список

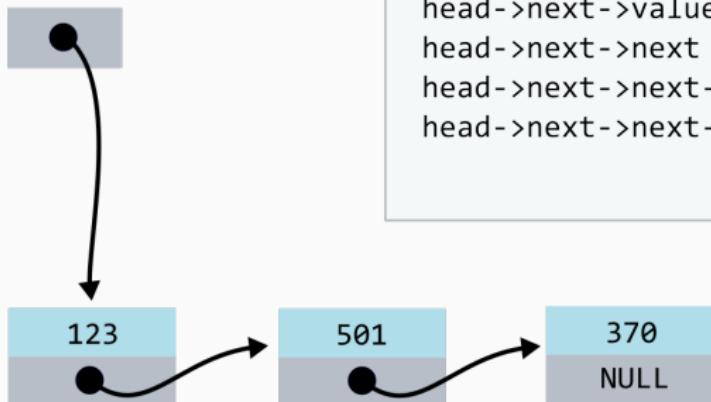
head



```
Node* head = malloc(sizeof(Node));  
head->value = 123;  
head->next = malloc(sizeof(Node));  
head->next->value = 501;  
head->next->next = malloc(sizeof(Node));
```

# Связный список

head



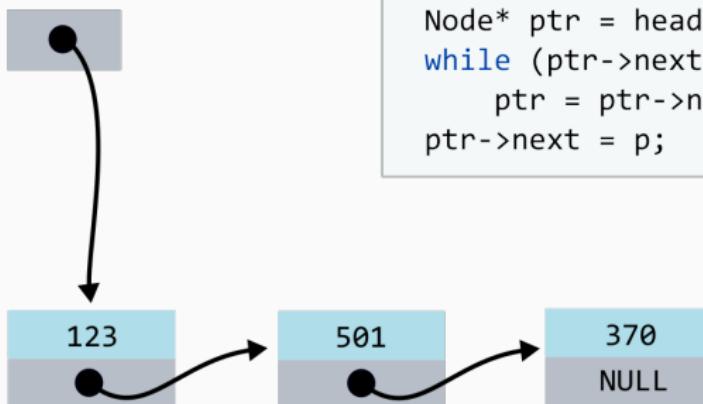
```
Node* head = malloc(sizeof(Node));  
head->value = 123;  
head->next = malloc(sizeof(Node));  
head->next->value = 501;  
head->next->next = malloc(sizeof(Node));  
head->next->next->value = 370;  
head->next->next->next = NULL;
```

## **Добавление элемента в связный список**

---

# Добавление элемента в связный список

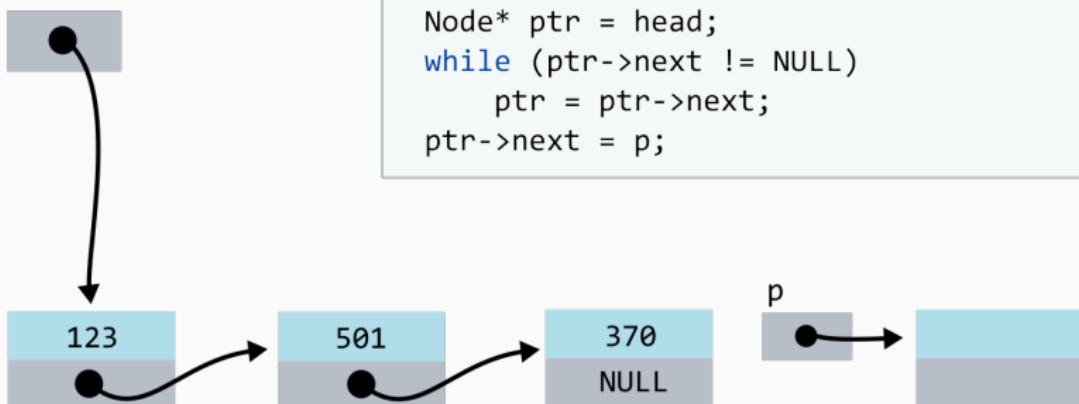
head



```
Node* p = malloc(sizeof(Node));  
p->value = 100;  
p->next = NULL;  
  
Node* ptr = head;  
while (ptr->next != NULL)  
    ptr = ptr->next;  
ptr->next = p;
```

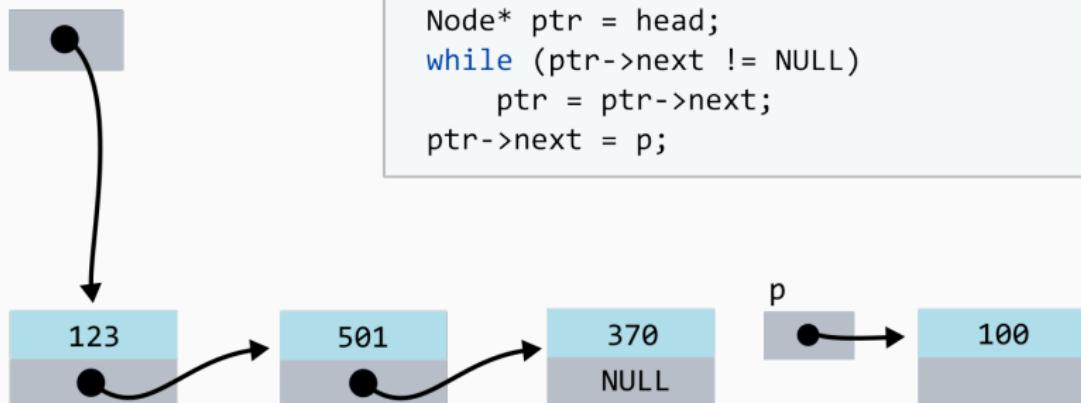
# Добавление элемента в связный список

head



# Добавление элемента в связный список

head



# Добавление элемента в связный список

head

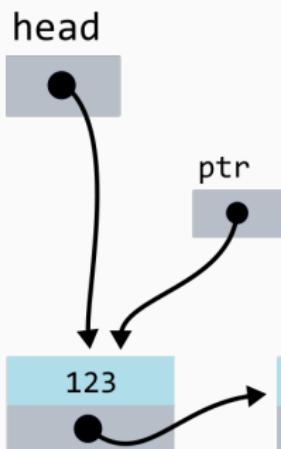


```
Node* p = malloc(sizeof(Node));  
p->value = 100;  
p->next = NULL;
```

```
Node* ptr = head;  
while (ptr->next != NULL)  
    ptr = ptr->next;  
ptr->next = p;
```



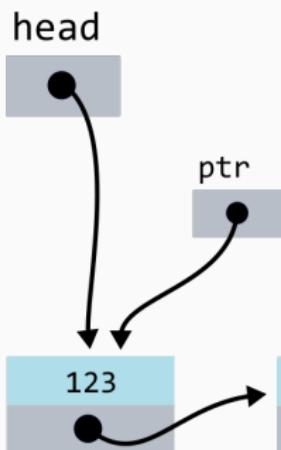
# Добавление элемента в связный список



```
Node* p = malloc(sizeof(Node));  
p->value = 100;  
p->next = NULL;
```

```
Node* ptr = head;  
while (ptr->next != NULL)  
    ptr = ptr->next;  
ptr->next = p;
```

# Добавление элемента в связный список

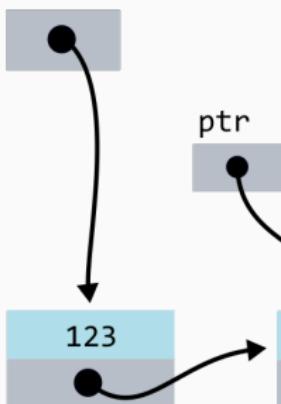


```
Node* p = malloc(sizeof(Node));  
p->value = 100;  
p->next = NULL;
```

```
Node* ptr = head;  
while (ptr->next != NULL)  
    ptr = ptr->next;  
ptr->next = p;
```

# Добавление элемента в связный список

head



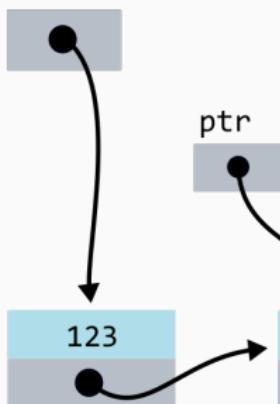
```
Node* p = malloc(sizeof(Node));  
p->value = 100;  
p->next = NULL;
```

```
Node* ptr = head;  
while (ptr->next != NULL)  
    ptr = ptr->next;  
ptr->next = p;
```



# Добавление элемента в связный список

head



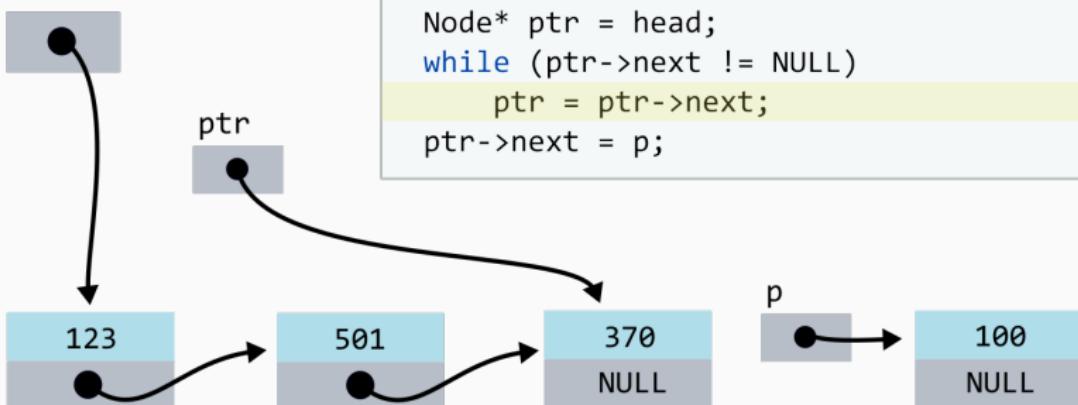
```
Node* p = malloc(sizeof(Node));  
p->value = 100;  
p->next = NULL;
```

```
Node* ptr = head;  
while (ptr->next != NULL)  
    ptr = ptr->next;  
ptr->next = p;
```



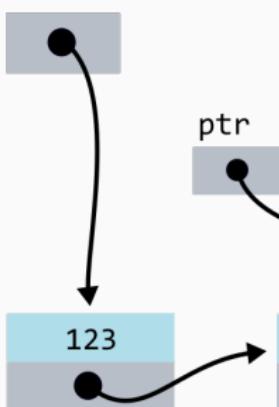
# Добавление элемента в связный список

head



# Добавление элемента в связный список

head



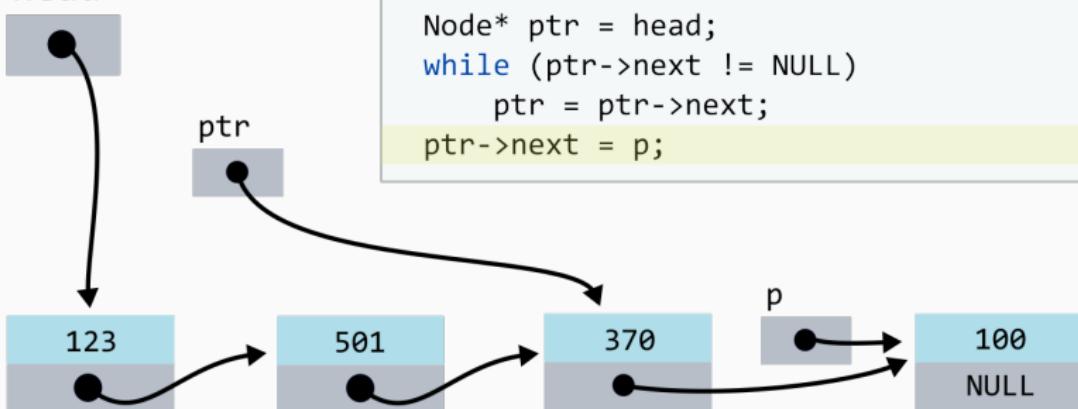
```
Node* p = malloc(sizeof(Node));  
p->value = 100;  
p->next = NULL;
```

```
Node* ptr = head;  
while (ptr->next != NULL)  
    ptr = ptr->next;  
ptr->next = p;
```



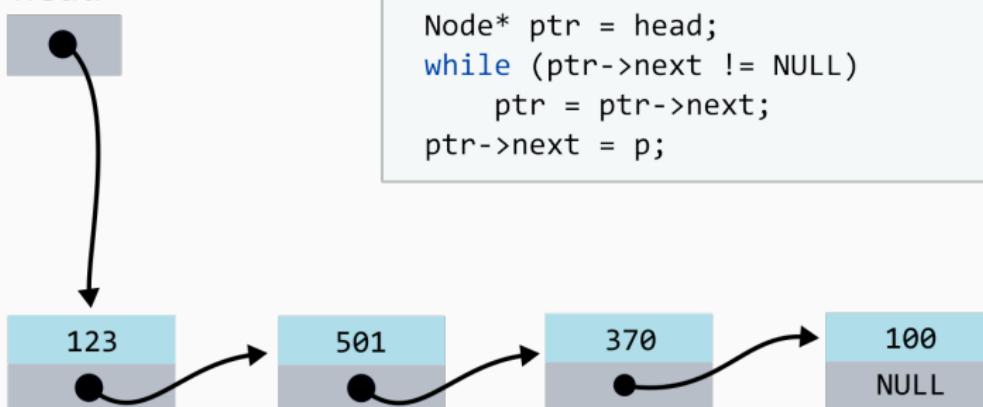
# Добавление элемента в связный список

head



# Добавление элемента в связный список

head



## **Двусвязный список**

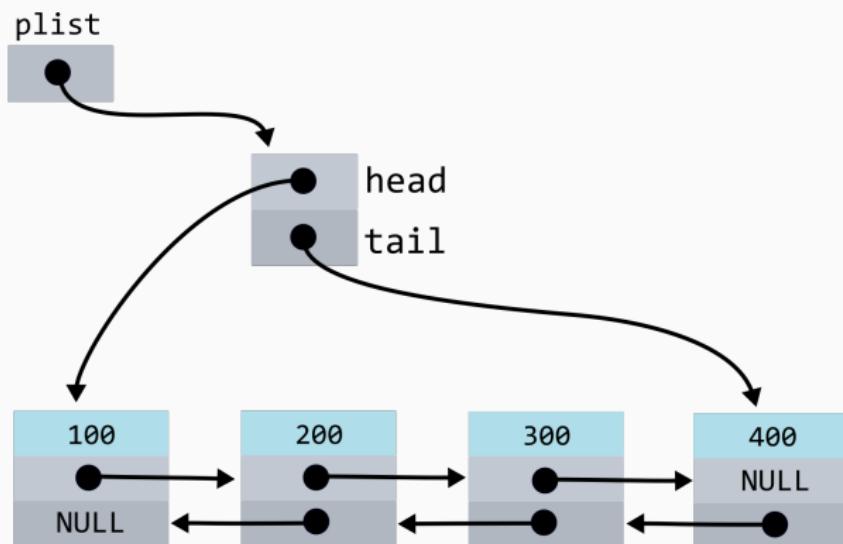
---

# Двусвязный список



```
struct Node
{
    int value;
    Node* next;
    Node* prev;
};
```

## Двусвязный список



# **Связные списки в стандартной библиотеке C++. std::list и std::forward\_list**

---

- std::forward\_list - это односвязный список

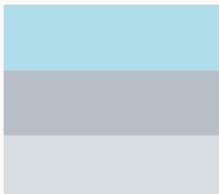
```
std::forward_list a {100, 200, 300, 400};  
a.push_front(500);
```

- std::list - это двусвязный список

```
std::list a {100, 200, 300, 400};  
a.push_front(500);  
a.push_back(500);
```

## Бинарное дерево

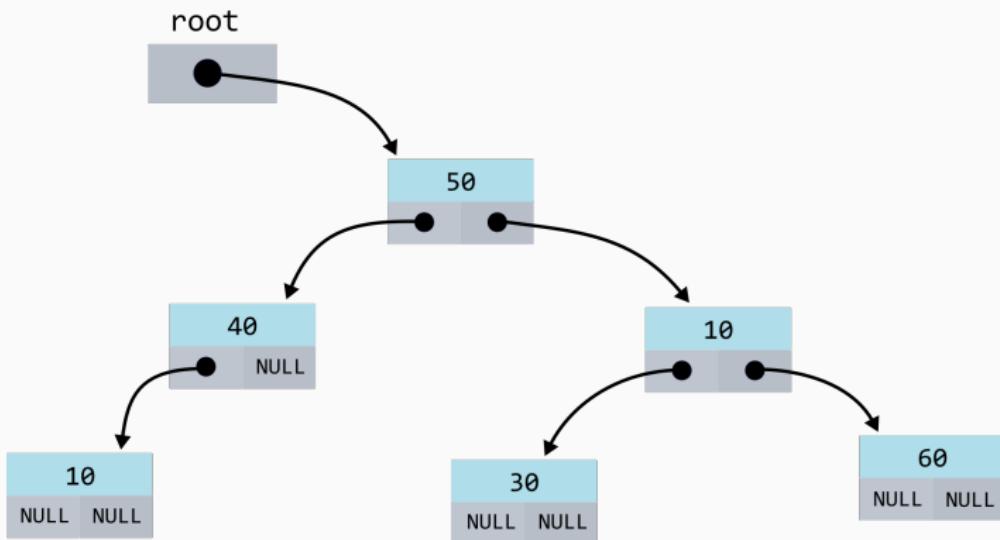
---



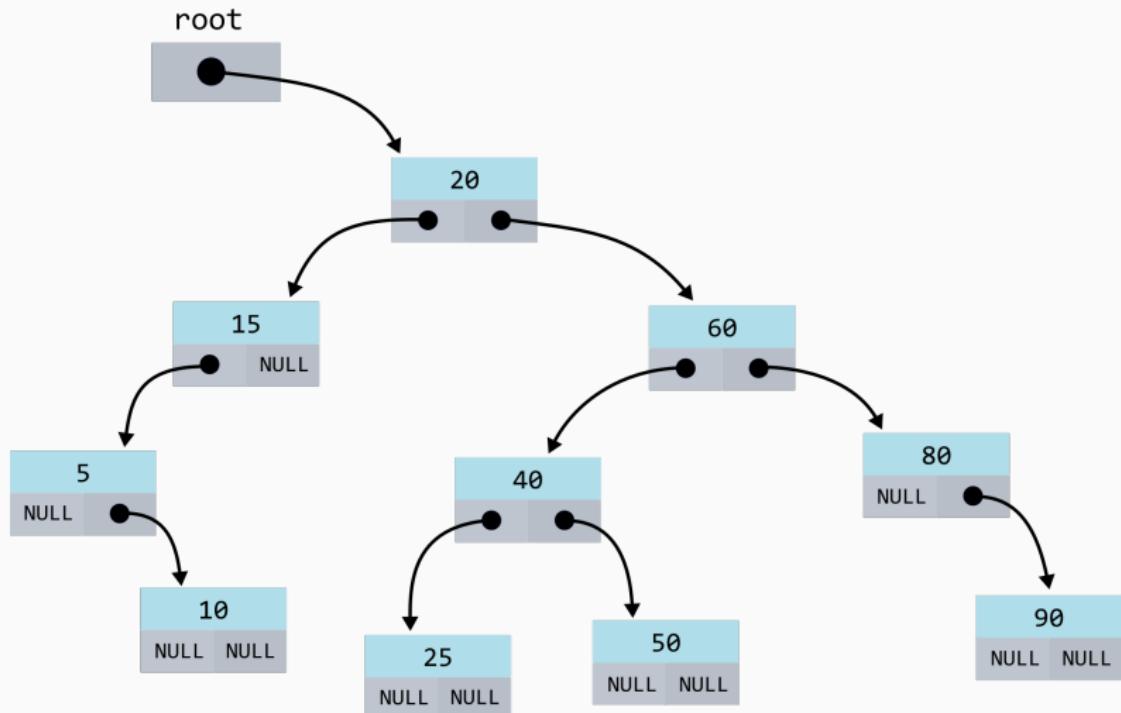
value  
left  
right

```
struct Node
{
    int value;
    Node* left;
    Node* right;
};
```

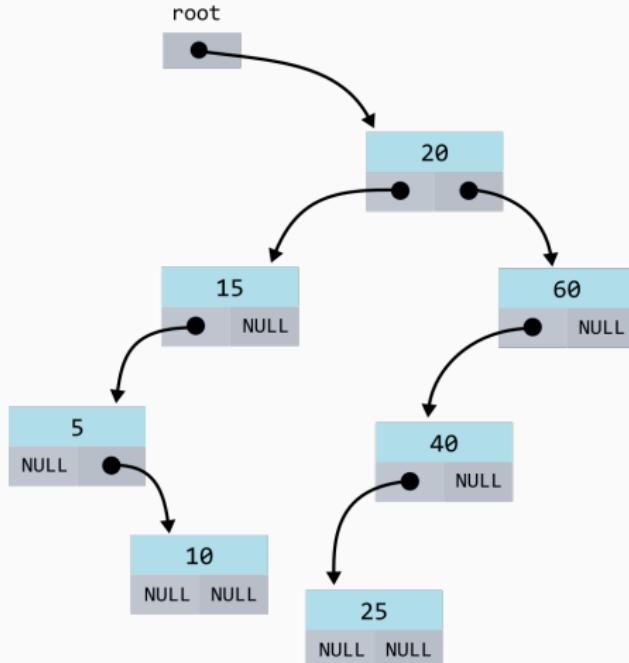
# Бинарное дерево



# Бинарное дерево поиска

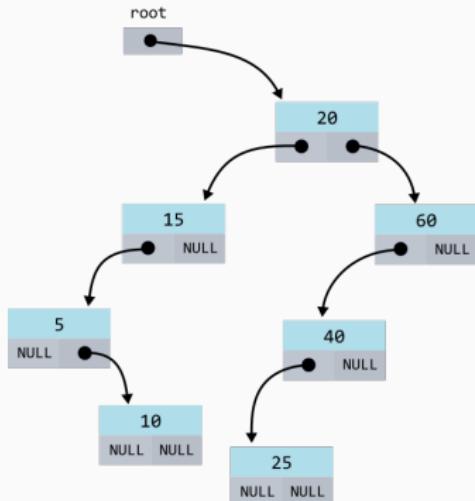


# Вставка элемента в бинарное дерево поиска



# Вставка элемента в бинарное дерево поиска

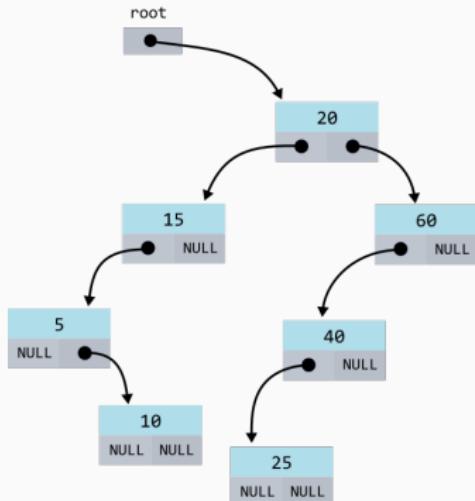
bst\_insert(root, 50)



```
Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
    {
        root->left = bst_insert(root->left, x);
    }
    else if (x > root->value)
    {
        root->right = bst_insert(root->right, x);
    }
    return root;
}
```

# Вставка элемента в бинарное дерево поиска

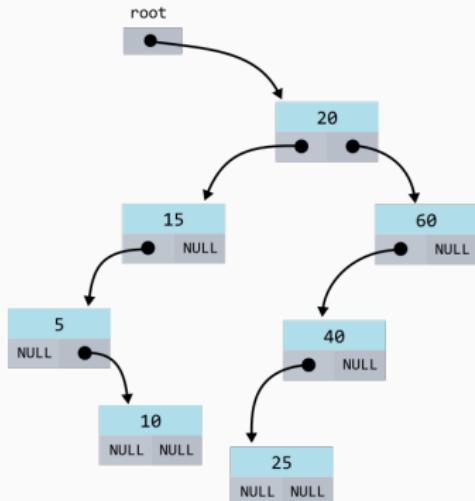
```
bst_insert(root, 50)
```



```
Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
    {
        root->left = bst_insert(root->left, x);
    }
    else if (x > root->value)
    {
        root->right = bst_insert(root->right, x);
    }
    return root;
}
```

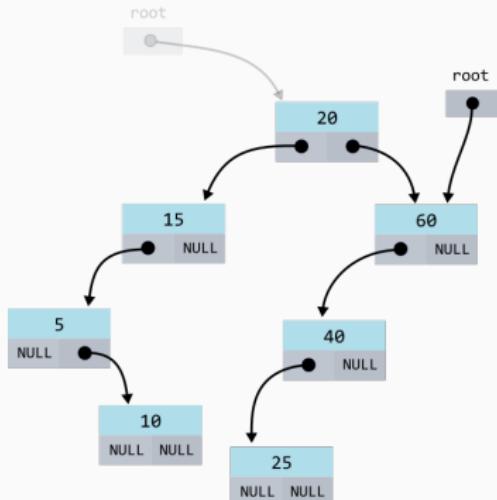
# Вставка элемента в бинарное дерево поиска

```
bst_insert(root, 50)
```



```
Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
    {
        root->left = bst_insert(root->left, x);
    }
    else if (x > root->value)
    {
        root->right = bst_insert(root->right, x);
    }
    return root;
}
```

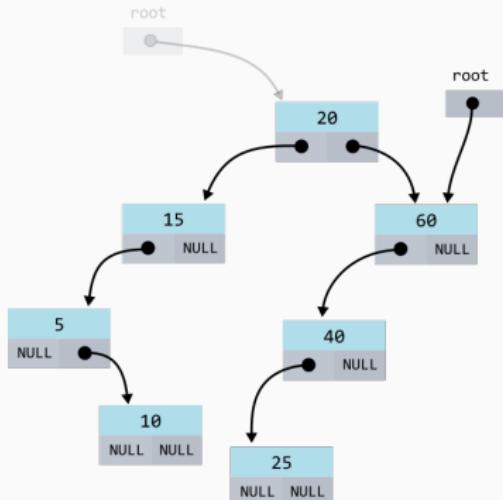
# Вставка элемента в бинарное дерево поиска



```
bst_insert(root, 50)  
bst_insert(root->right, 50)
```

```
Node* bst_insert(Node* root, int x)  
{  
    if (root == NULL)  
    {  
        root = (Node*)malloc(sizeof(Node));  
        root->value = x;  
        root->left = NULL;  
        root->right = NULL;  
    }  
    else if (x < root->value)  
    {  
        root->left = bst_insert(root->left, x);  
    }  
    else if (x > root->value)  
    {  
        root->right = bst_insert(root->right, x);  
    }  
    return root;  
}
```

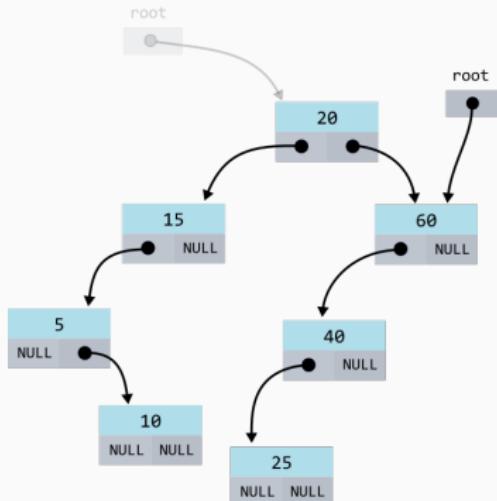
# Вставка элемента в бинарное дерево поиска



```
bst_insert(root, 50)
bst_insert(root->right, 50)
```

```
Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
    {
        root->left = bst_insert(root->left, x);
    }
    else if (x > root->value)
    {
        root->right = bst_insert(root->right, x);
    }
    return root;
}
```

# Вставка элемента в бинарное дерево поиска

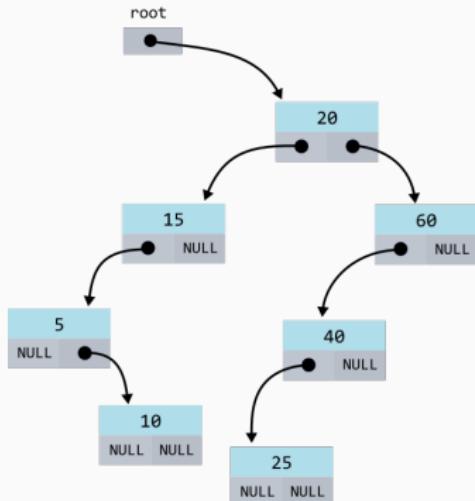


```
bst_insert(root, 50)  
bst_insert(root->right, 50)
```

```
Node* bst_insert(Node* root, int x)  
{  
    if (root == NULL)  
    {  
        root = (Node*)malloc(sizeof(Node));  
        root->value = x;  
        root->left = NULL;  
        root->right = NULL;  
    }  
    else if (x < root->value)  
    {  
        root->left = bst_insert(root->left, x);  
    }  
    else if (x > root->value)  
    {  
        root->right = bst_insert(root->right, x);  
    }  
    return root;  
}
```

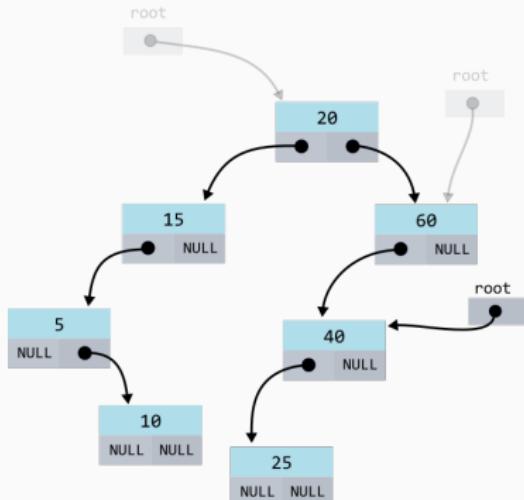
# Вставка элемента в бинарное дерево поиска

```
bst_insert(root, 50)
```



```
Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
    {
        root->left = bst_insert(root->left, x);
    }
    else if (x > root->value)
    {
        root->right = bst_insert(root->right, x);
    }
    return root;
}
```

# Вставка элемента в бинарное дерево поиска

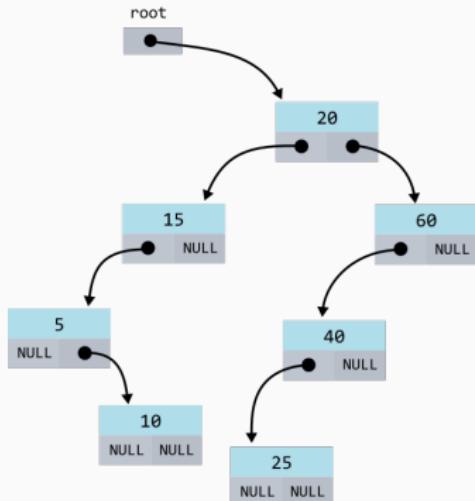


```
bst_insert(root, 50)
bst_insert(root->right, 50)
bst_insert(root->right->left, 50)
```

```
Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
    {
        root->left = bst_insert(root->left, x);
    }
    else if (x > root->value)
    {
        root->right = bst_insert(root->right, x);
    }
    return root;
}
```

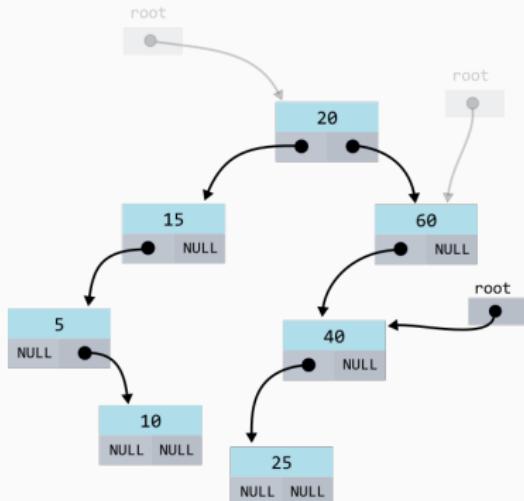
# Вставка элемента в бинарное дерево поиска

`bst_insert(root, 50)`



```
Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
    {
        root->left = bst_insert(root->left, x);
    }
    else if (x > root->value)
    {
        root->right = bst_insert(root->right, x);
    }
    return root;
}
```

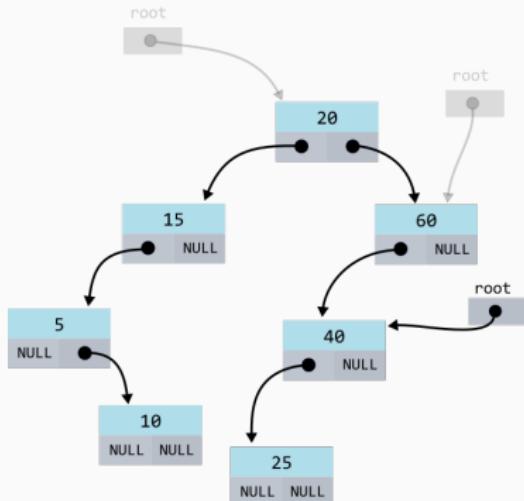
# Вставка элемента в бинарное дерево поиска



```
bst_insert(root, 50)
bst_insert(root->right, 50)
bst_insert(root->right->left, 50)
```

```
Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
    {
        root->left = bst_insert(root->left, x);
    }
    else if (x > root->value)
    {
        root->right = bst_insert(root->right, x);
    }
    return root;
}
```

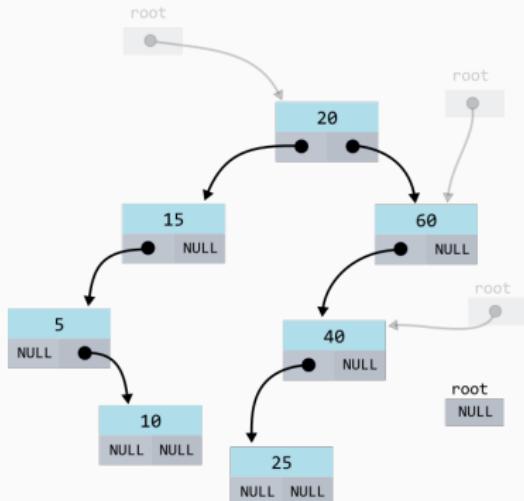
# Вставка элемента в бинарное дерево поиска



```
bst_insert(root, 50)
bst_insert(root->right, 50)
bst_insert(root->right->left, 50)
```

```
Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
    {
        root->left = bst_insert(root->left, x);
    }
    else if (x > root->value)
    {
        root->right = bst_insert(root->right, x);
    }
    return root;
}
```

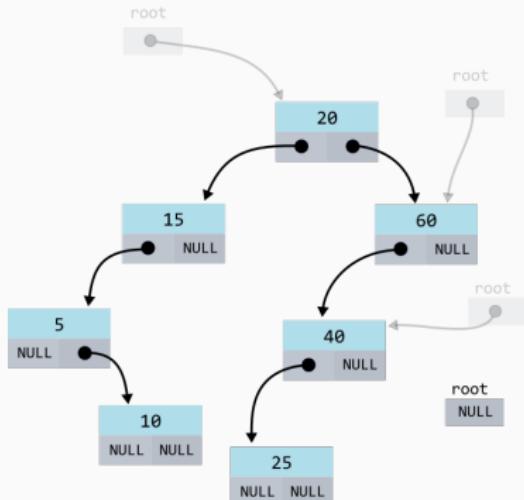
# Вставка элемента в бинарное дерево поиска



```
bst_insert(root, 50)
bst_insert(root->right, 50)
bst_insert(root->right->left, 50)
bst_insert(NULL, 50)
```

```
Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
    {
        root->left = bst_insert(root->left, x);
    }
    else if (x > root->value)
    {
        root->right = bst_insert(root->right, x);
    }
    return root;
}
```

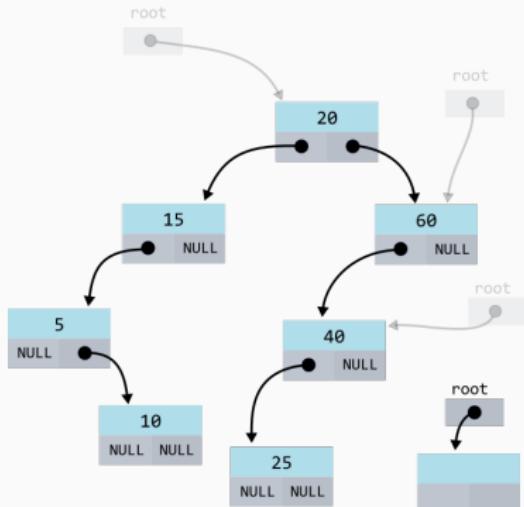
# Вставка элемента в бинарное дерево поиска



```
bst_insert(root, 50)
bst_insert(root->right, 50)
bst_insert(root->right->left, 50)
bst_insert(NULL, 50)
```

```
Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
    {
        root->left = bst_insert(root->left, x);
    }
    else if (x > root->value)
    {
        root->right = bst_insert(root->right, x);
    }
    return root;
}
```

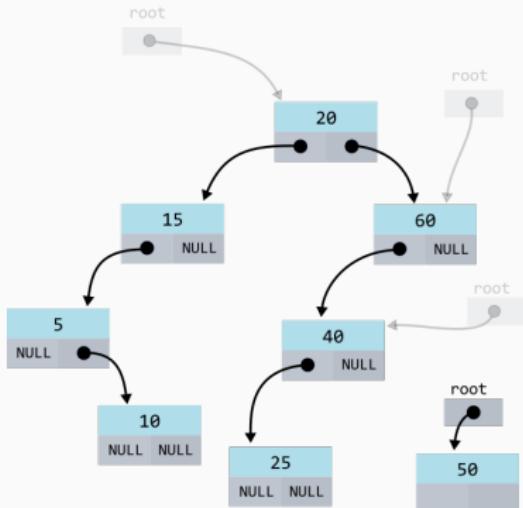
# Вставка элемента в бинарное дерево поиска



```
bst_insert(root, 50)
bst_insert(root->right, 50)
bst_insert(root->right->left, 50)
bst_insert(NULL, 50)
```

```
Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
    {
        root->left = bst_insert(root->left, x);
    }
    else if (x > root->value)
    {
        root->right = bst_insert(root->right, x);
    }
    return root;
}
```

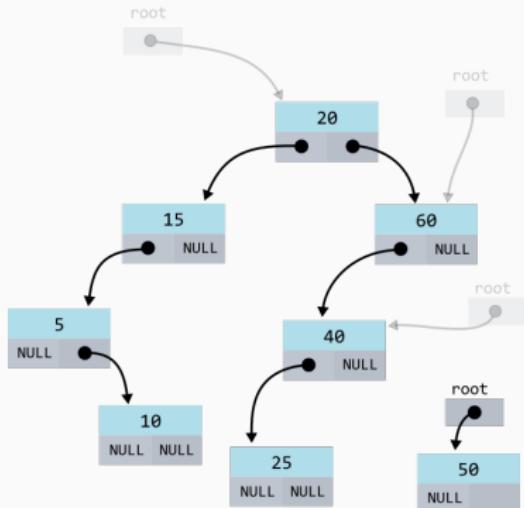
# Вставка элемента в бинарное дерево поиска



```
bst_insert(root, 50)
bst_insert(root->right, 50)
bst_insert(root->right->left, 50)
bst_insert(NULL, 50)
```

```
Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
    {
        root->left = bst_insert(root->left, x);
    }
    else if (x > root->value)
    {
        root->right = bst_insert(root->right, x);
    }
    return root;
}
```

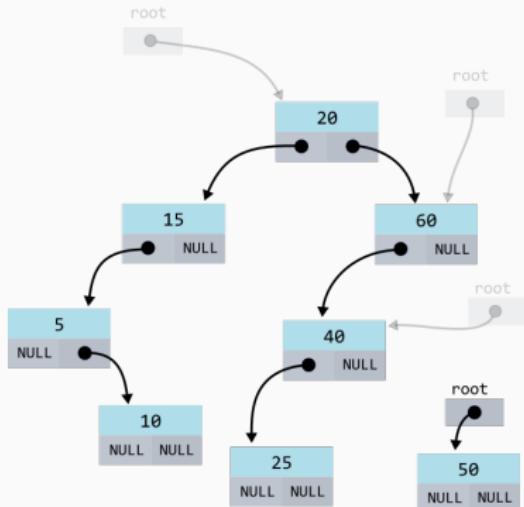
# Вставка элемента в бинарное дерево поиска



```
bst_insert(root, 50)
bst_insert(root->right, 50)
bst_insert(root->right->left, 50)
bst_insert(NULL, 50)
```

```
Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
    {
        root->left = bst_insert(root->left, x);
    }
    else if (x > root->value)
    {
        root->right = bst_insert(root->right, x);
    }
    return root;
}
```

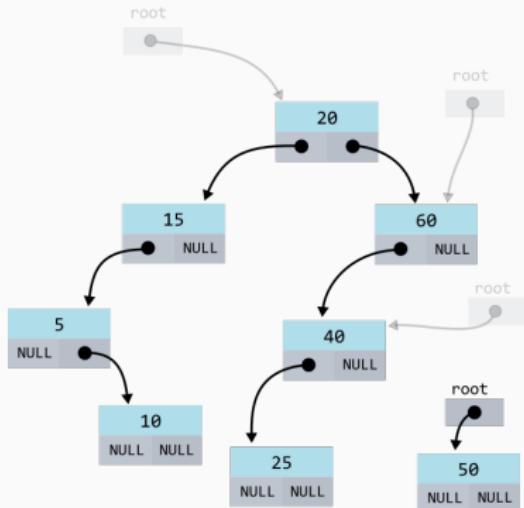
# Вставка элемента в бинарное дерево поиска



```
bst_insert(root, 50)
bst_insert(root->right, 50)
bst_insert(root->right->left, 50)
bst_insert(NULL, 50)
```

```
Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
    {
        root->left = bst_insert(root->left, x);
    }
    else if (x > root->value)
    {
        root->right = bst_insert(root->right, x);
    }
    return root;
}
```

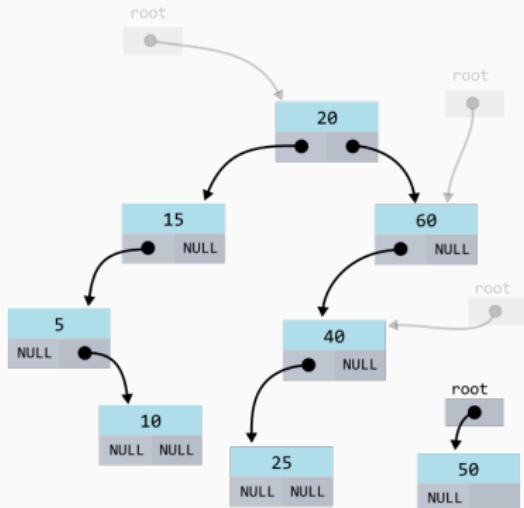
# Вставка элемента в бинарное дерево поиска



```
bst_insert(root, 50)
bst_insert(root->right, 50)
bst_insert(root->right->left, 50)
bst_insert(NULL, 50)
```

```
Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
    {
        root->left = bst_insert(root->left, x);
    }
    else if (x > root->value)
    {
        root->right = bst_insert(root->right, x);
    }
    return root;
}
```

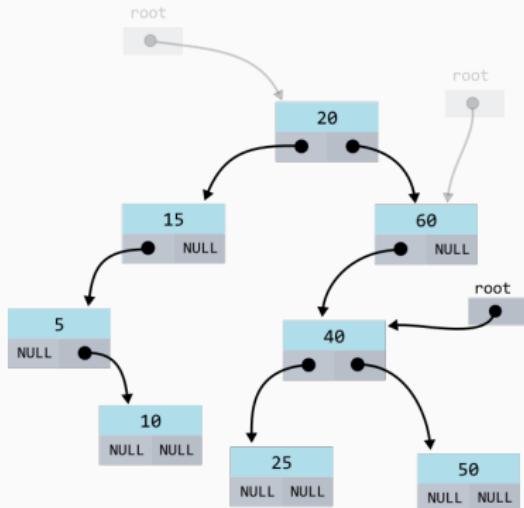
# Вставка элемента в бинарное дерево поиска



```
bst_insert(root, 50)
bst_insert(root->right, 50)
bst_insert(root->right->left, 50)
bst_insert(NULL, 50)
```

```
Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
    {
        root->left = bst_insert(root->left, x);
    }
    else if (x > root->value)
    {
        root->right = bst_insert(root->right, x);
    }
    return root;
}
```

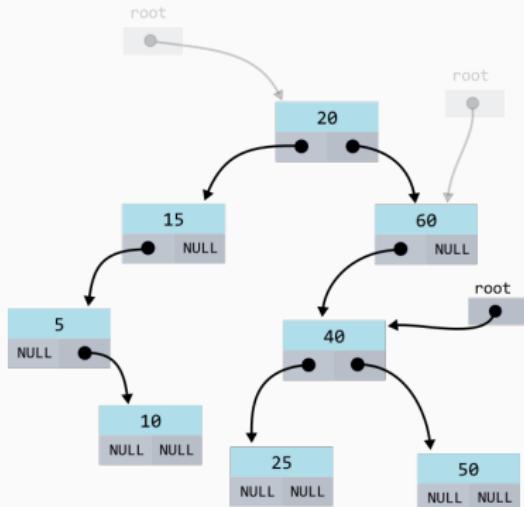
# Вставка элемента в бинарное дерево поиска



```
bst_insert(root, 50)
bst_insert(root->right, 50)
bst_insert(root->right->left, 50)
```

```
Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
    {
        root->left = bst_insert(root->left, x);
    }
    else if (x > root->value)
    {
        root->right = bst_insert(root->right, x);
    }
    return root;
}
```

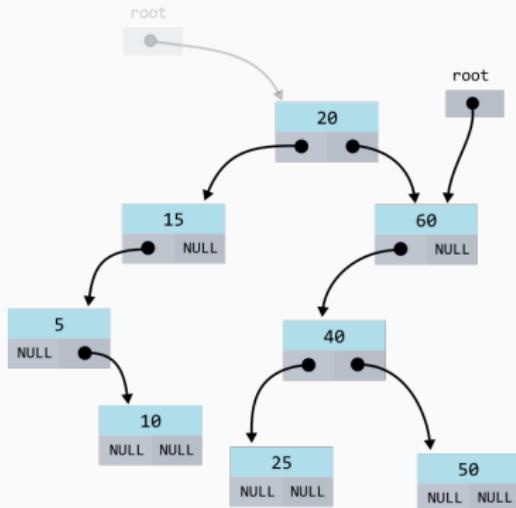
# Вставка элемента в бинарное дерево поиска



```
bst_insert(root, 50)
bst_insert(root->right, 50)
bst_insert(root->right->left, 50)
```

```
Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
    {
        root->left = bst_insert(root->left, x);
    }
    else if (x > root->value)
    {
        root->right = bst_insert(root->right, x);
    }
    return root;
}
```

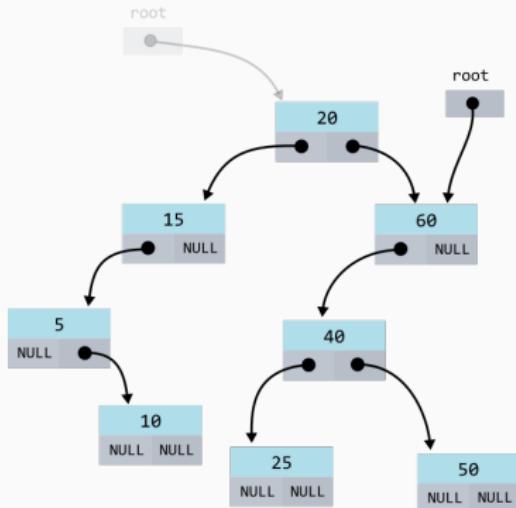
# Вставка элемента в бинарное дерево поиска



```
bst_insert(root, 50)
bst_insert(root->right, 50)
```

```
Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
    {
        root->left = bst_insert(root->left, x);
    }
    else if (x > root->value)
    {
        root->right = bst_insert(root->right, x);
    }
    return root;
}
```

# Вставка элемента в бинарное дерево поиска

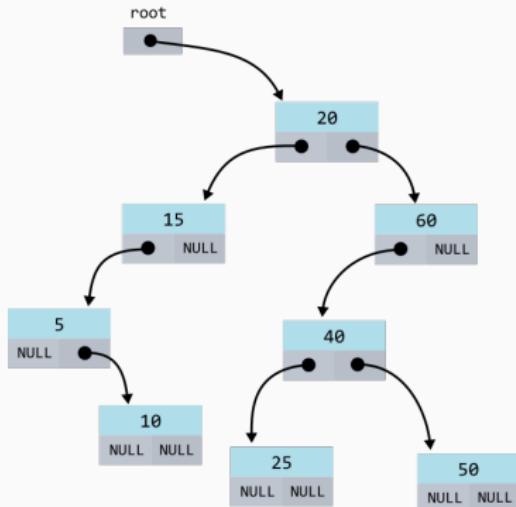


```
bst_insert(root, 50)  
bst_insert(root->right, 50)
```

```
Node* bst_insert(Node* root, int x)  
{  
    if (root == NULL)  
    {  
        root = (Node*)malloc(sizeof(Node));  
        root->value = x;  
        root->left = NULL;  
        root->right = NULL;  
    }  
    else if (x < root->value)  
    {  
        root->left = bst_insert(root->left, x);  
    }  
    else if (x > root->value)  
    {  
        root->right = bst_insert(root->right, x);  
    }  
    return root;  
}
```

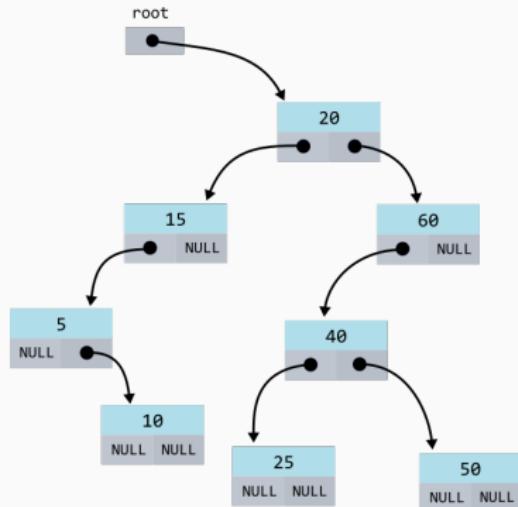
# Вставка элемента в бинарное дерево поиска

bst\_insert(root, 50)



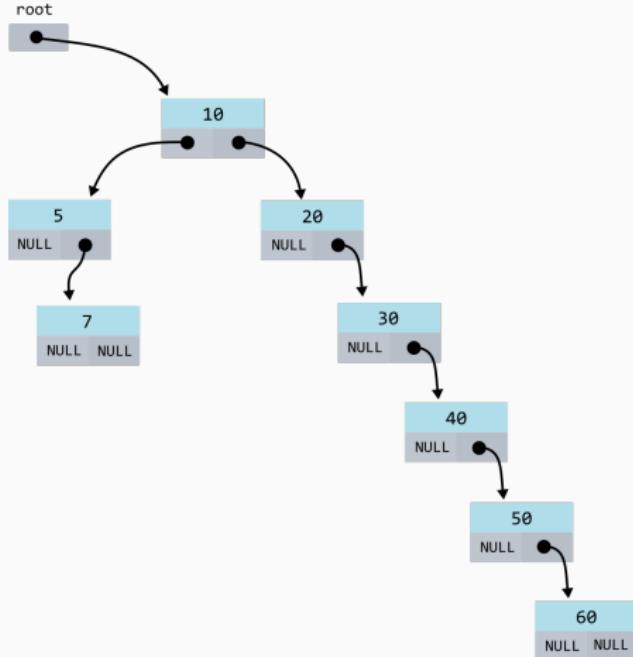
```
Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
    {
        root->left = bst_insert(root->left, x);
    }
    else if (x > root->value)
    {
        root->right = bst_insert(root->right, x);
    }
    return root;
}
```

# Вставка элемента в бинарное дерево поиска

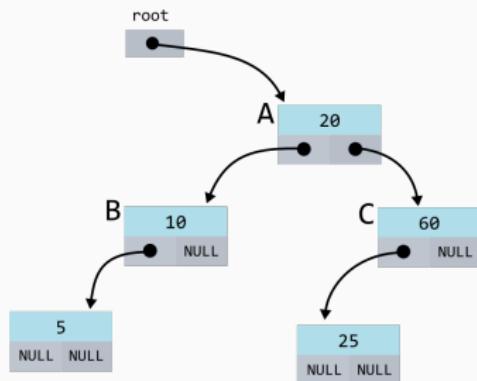


```
Node* bst_insert(Node* root, int x)
{
    if (root == NULL)
    {
        root = (Node*)malloc(sizeof(Node));
        root->value = x;
        root->left = NULL;
        root->right = NULL;
    }
    else if (x < root->value)
    {
        root->left = bst_insert(root->left, x);
    }
    else if (x > root->value)
    {
        root->right = bst_insert(root->right, x);
    }
    return root;
}
```

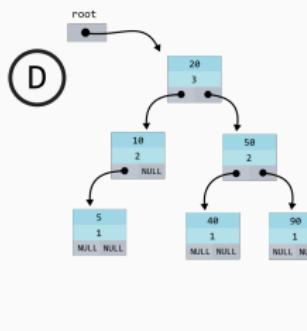
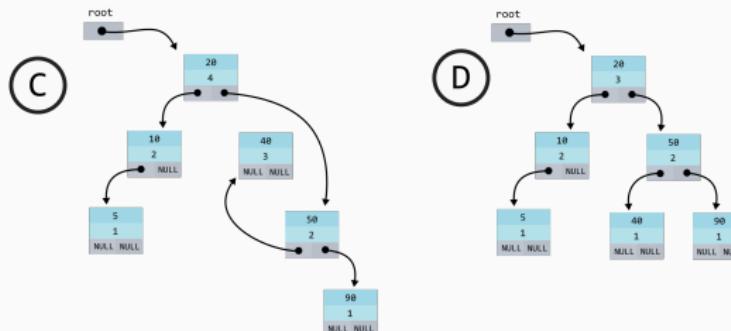
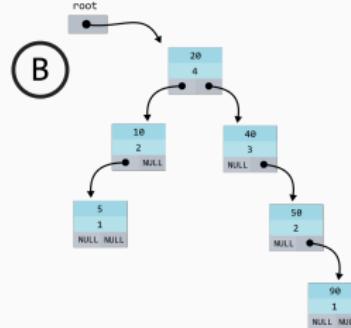
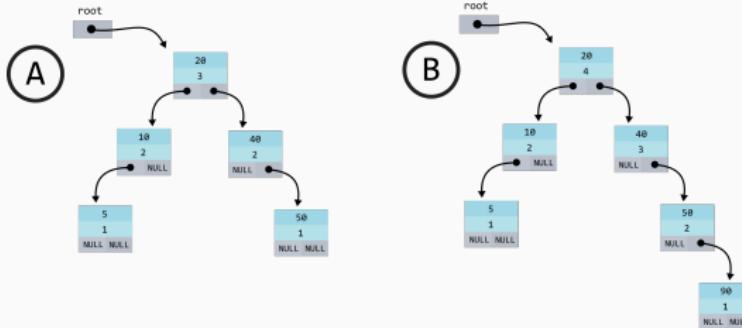
# Несбалансированное дерево поиска



# Сбалансированное дерево поиска



# Самобалансирующееся дерево поиска (AVL)



# Дерево поиска в стандартной библиотеке C++.

---

## Множества std::set и std::multiset

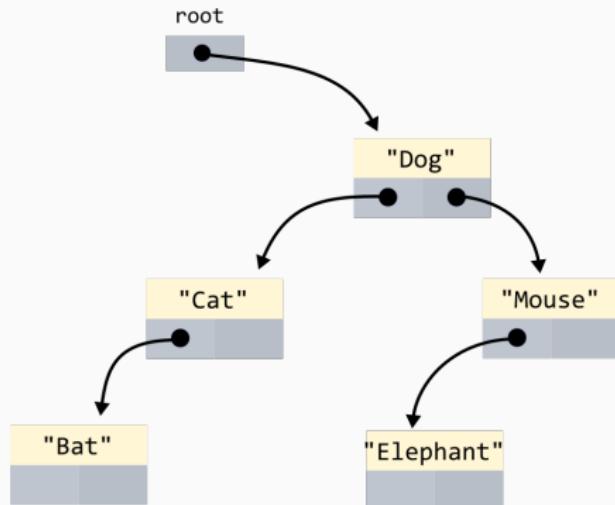
- std::set - это упорядоченное множество (как правило реализовано с помощью самобалансирующегося дерева поиска)

```
std::set a {10, 20, 30, 40, 50};  
a.insert(60);
```

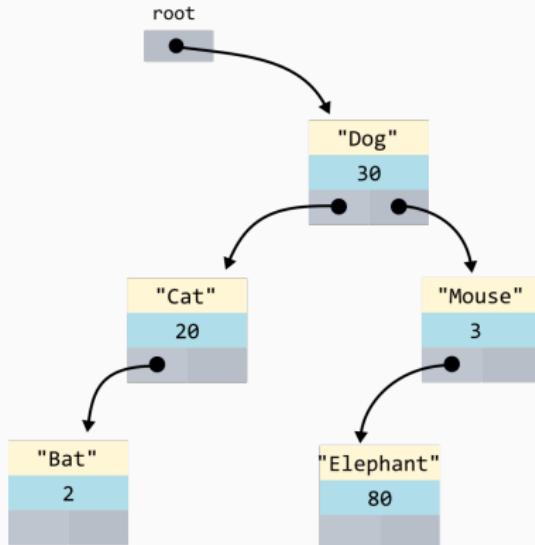
- std::multiset - это упорядоченное множество, которое может хранить дубликаты.

```
std::multiset a {10, 20, 10, 20, 50};  
a.insert(50);
```

```
std::set<std::string> s {"Cat", "Mouse", "Dog"};
s.insert("Bat");
s.insert("Elephant");
```



```
std::map<std::string, int> m{{"Cat", 20}, {"Mouse", 3}};  
m["Dog"] = 30;  
m["Bat"] = 2;  
m["Elephant"] = 80;
```



## Хеш-таблица

---

## Множество std::unordered\_set

- std::unordered\_set - это неупорядоченное множество (как правило реализовано с помощью хеш-таблицы)

```
std::unordered_set a {10, 20, 30, 40, 50};  
a.insert(60);
```

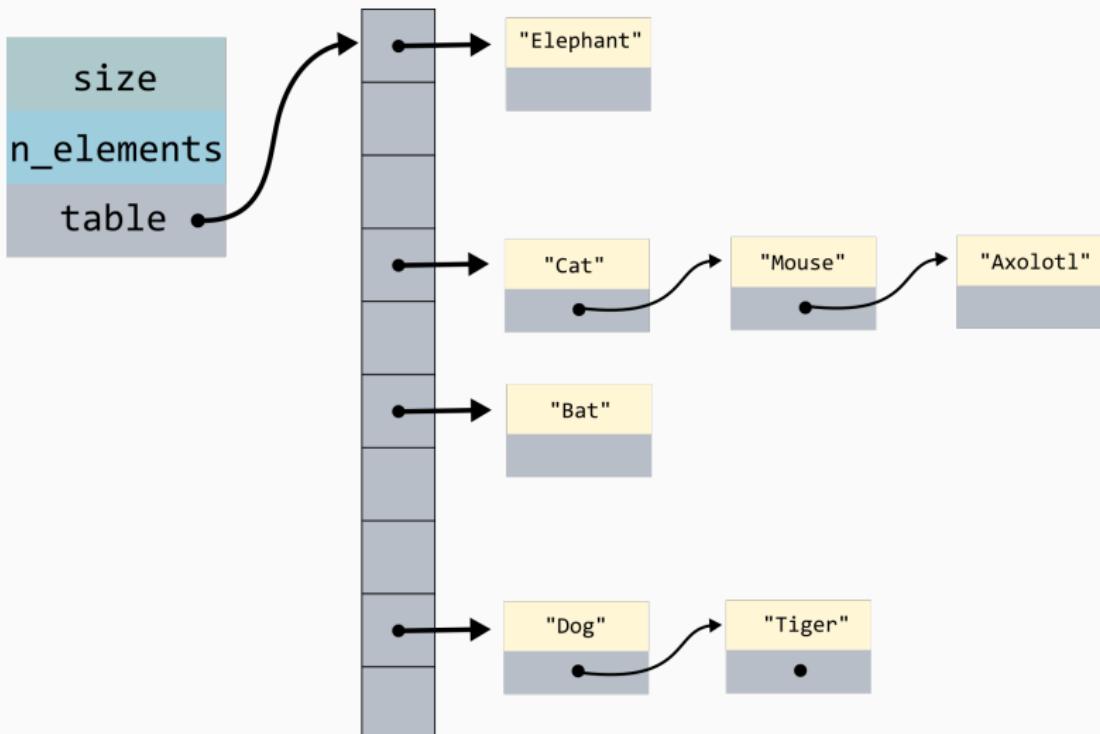
- std::unordered\_multiset - это неупорядоченное множество (как правило реализованное с помощью хеш-таблицы), которое может хранить дубликаты.

```
std::unordered_multiset a {10, 20, 10, 20, 50};  
a.insert(50);
```

## Множество std::unordered\_set

```
std::unordered_set<std::string> s {"Cat", "Mouse"};
s.insert("Dog");
s.insert("Bat");
s.insert("Elephant");
s.insert("Axolotl");
s.insert("Tiger");
```

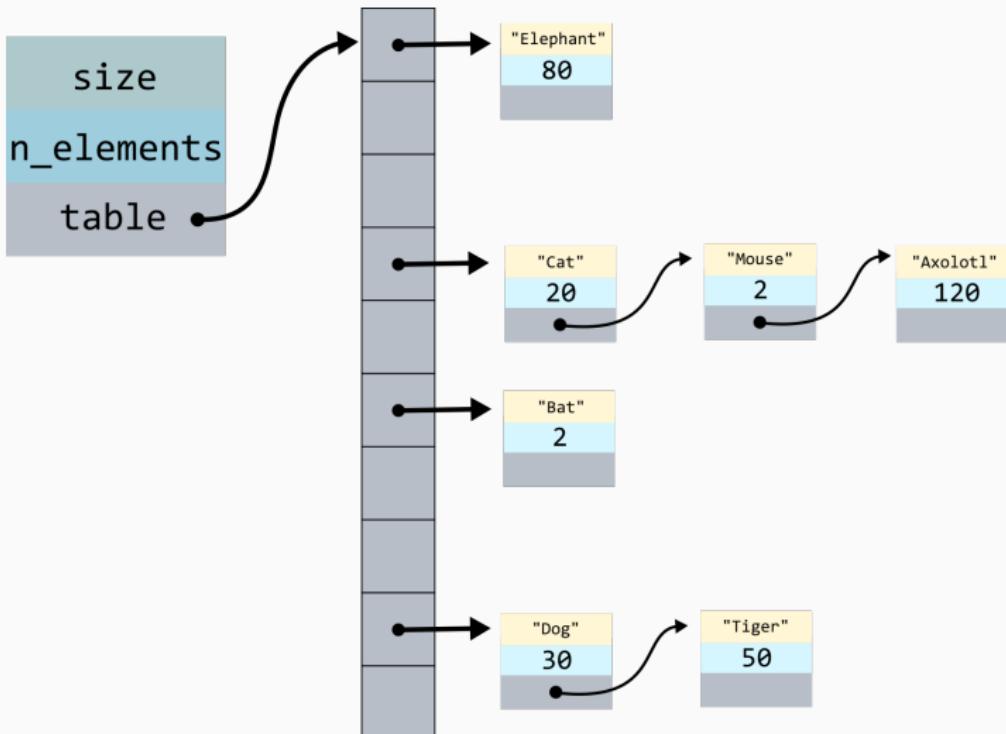
## Множество std::unordered\_set



## Словарь std::unordered\_map

```
std::unordered_map<std::string, int> m {{"Cat", 20},  
    m["Mouse"] = 2;  
    m["Dog"] = 30;  
    m["Bat"] = 2;  
    m["Elephant"] = 80;  
    m["Axolotl"] = 120;  
    m["Tiger"] = 50;
```

# Словарь std::unordered\_map



## Контейнеры

---

контейнер	описание и основные свойства
<code>std::array</code>	Массив фиксированного размера
<code>std::vector</code>	Динамический массив Все элементы лежат вплотную друг к другу Есть доступ по индексу за $O(1)$
<code>std::list</code>	Двусвязный список Вставка/удаление элементов за $O(1)$ если есть итератор на элемент
<code>std::forward_list</code>	Односвязный список Вставка/удаление элементов за $O(1)$ если есть итератор на предыдущий элемент

std::set	Реализация множества на основе сбалансированного дерева поиска. Хранит элементы без дубликатов, в отсортированном виде. Поиск/вставка/удаление элементов за $O(\log(N))$
std::map	Реализация словаря на основе сбалансированного дерева поиска. Хранит пары ключ-значения без дубликатов ключей, в отсортированном виде Поиск/вставка/удаление элементов за $O(\log(N))$

std::unordered_set	Реализация множества на основе хеш-таблицы Хранит элементы без дубликатов, в произвольном порядке Поиск/вставка/удаление элементов за $O(1)$ в среднем
std::unordered_map	Реализация словаря на основе хеш-таблицы Хранит пары ключ-значения без дубликатов ключей, в произвольном порядке Поиск/вставка/удаление элементов за $O(1)$ в среднем
std::multiset	То же самое, что std::set, но может хранить дублированные значения