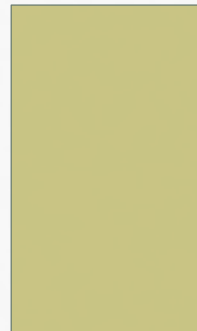


# Связные списки



# ТЕРМИНОЛОГИЯ

## Структуры данных

Массивы

статические

динамические

Списки

ОДНОСВЯЗНЫЕ

ДВУСВЯЗНЫЕ

Деревья

несбалансированные

АВЛ

красно-черные

Хэш-таблицы



вы здесь

Стек

Очередь

# ТЕРМИНОЛОГИЯ

## Контейнеры STL

[cplusplusreference.com](http://cplusplusreference.com)

Статья Обсуждение

C++ Библиотека контейнеров std::vector

**Последовательные**

std::array (C++11)  
vector  
deque  
forward\_list (C++11)  
list

**Ассоциативные**

set  
multiset  
map  
multimap

**Неупорядоченные ассоциативные**

unordered\_set (C++11)  
unordered\_multiset (C++11)  
unordered\_map (C++11)  
unordered\_multimap (C++11)

**Адаптеры**

1) stack  
2) queue  
priority\_queue

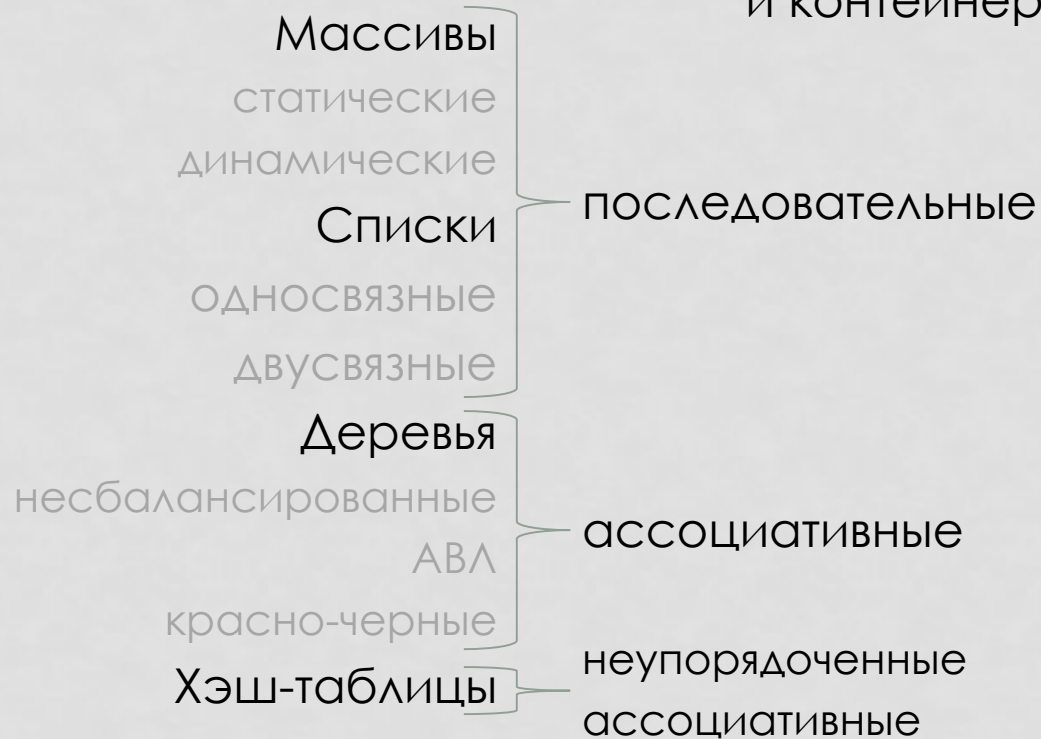
**Представления**

span (C++20)



# ТЕРМИНОЛОГИЯ

## Структуры данных и контейнеры

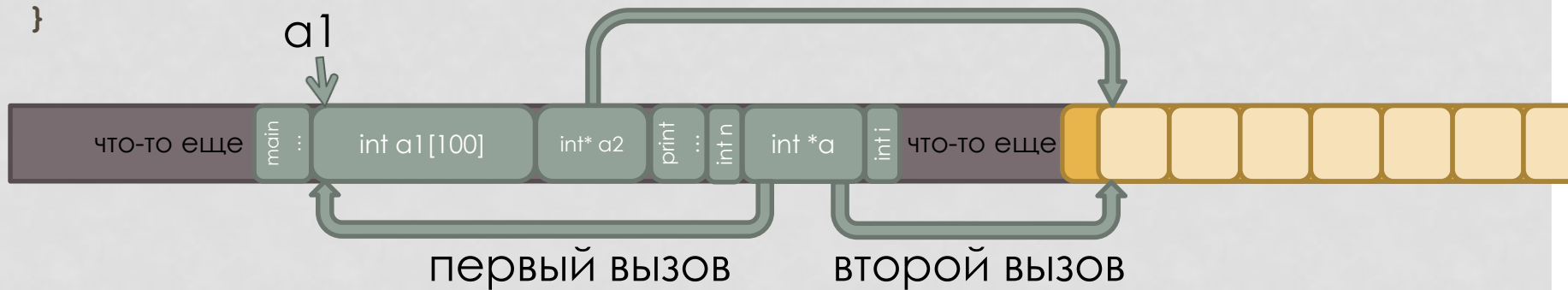


Стек  
Очередь



# СТАТИЧЕСКИЕ И ДИНАМИЧЕСКИЕ МАССИВЫ

```
void print(int n, int *a) {  
    for (int i = 0; i < n; i++)  
        cout << a[i] << " ";  
    cout << endl;  
}
```



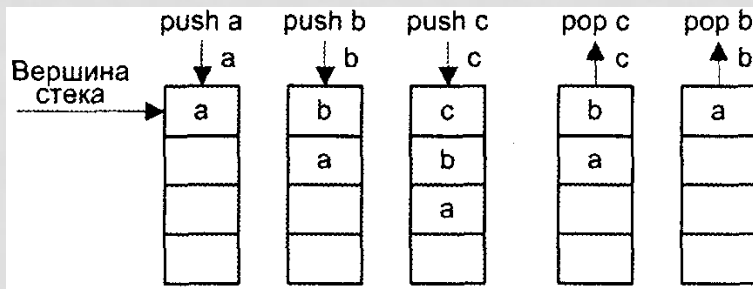
...

```
int a1[100], *a2 = new int[100];  
print(100, a1);  
print(100, a2);
```

# СТЕК И ОЧЕРЕДЬ

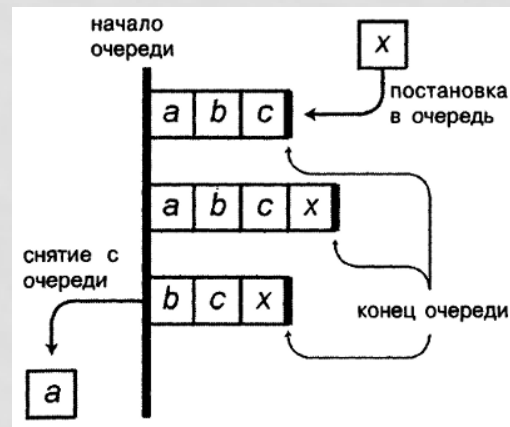
- Стек

- "последним вошел — первым вышел"
- last-in, first-out — LIFO
- top – начало стека
- push – запись
- pop – извлечение

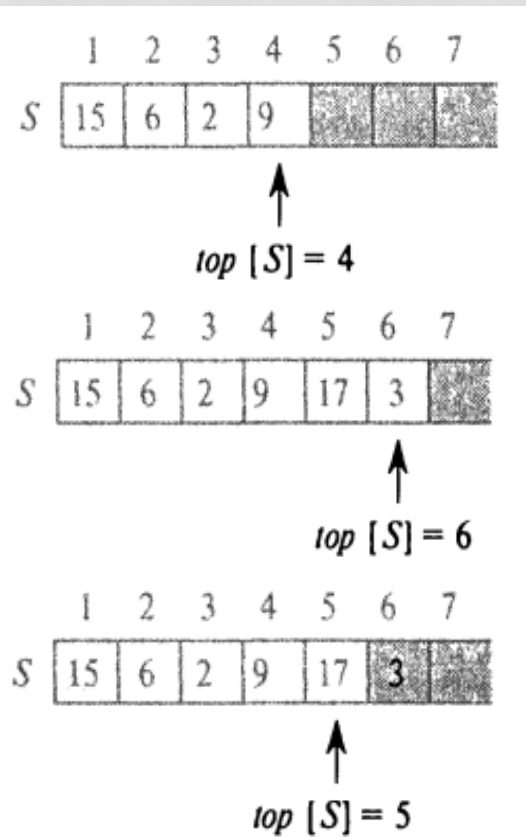


- Очередь

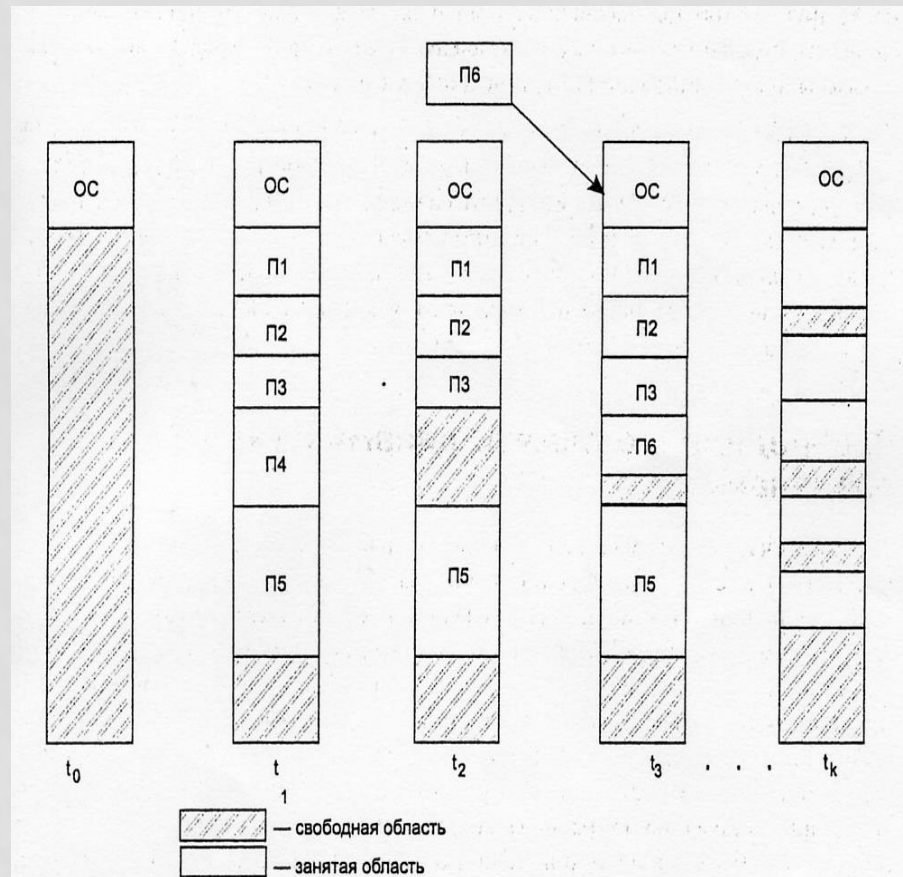
- "первым вошел — первым вышел"
- first-in, first-out — FIFO
- head – начало очереди
- tail – конец очереди
- enqueue – запись
- dequeue – извлечение



# ДОЛГОЕ ДОБАВЛЕНИЕ В НАЧАЛО/СЕРЕДИНУ



# ФРАГМЕНТАЦИЯ





# СВЯЗНЫЕ СПИСКИ

## Односвязные

- Singly linked list

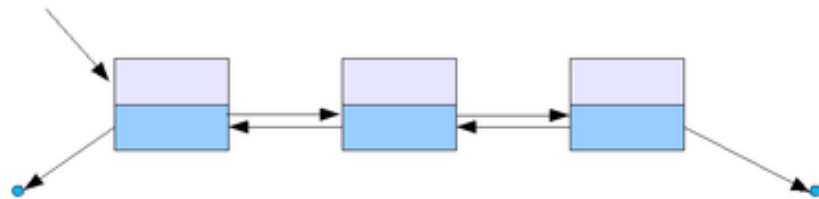
```
struct node_single {  
    int data;  
    node_single * next;  
};
```



## Двусвязные

- Doubly linked list

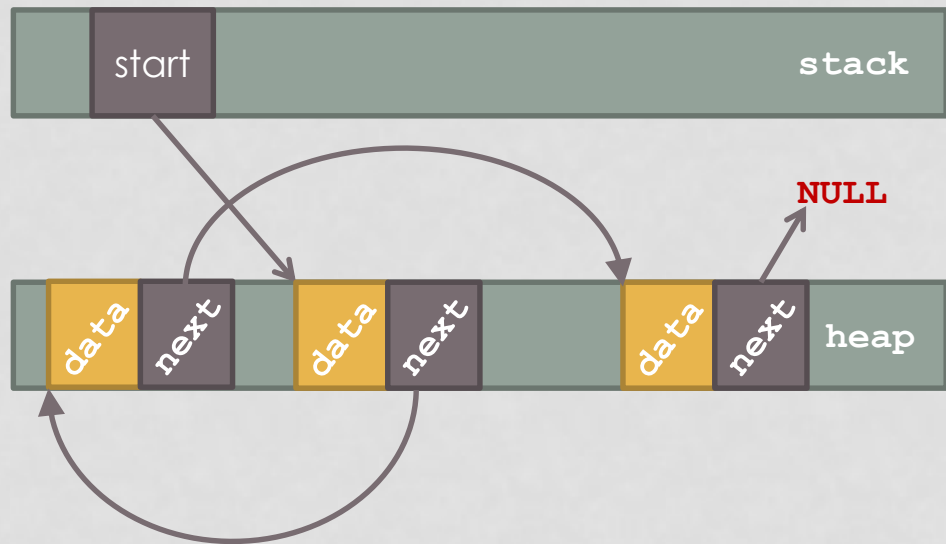
```
struct node_double {  
    int data;  
    node_double * next;  
    node_double * prev;  
};
```



```
struct node_single {  
    int data;  
    node_single * next;  
};
```

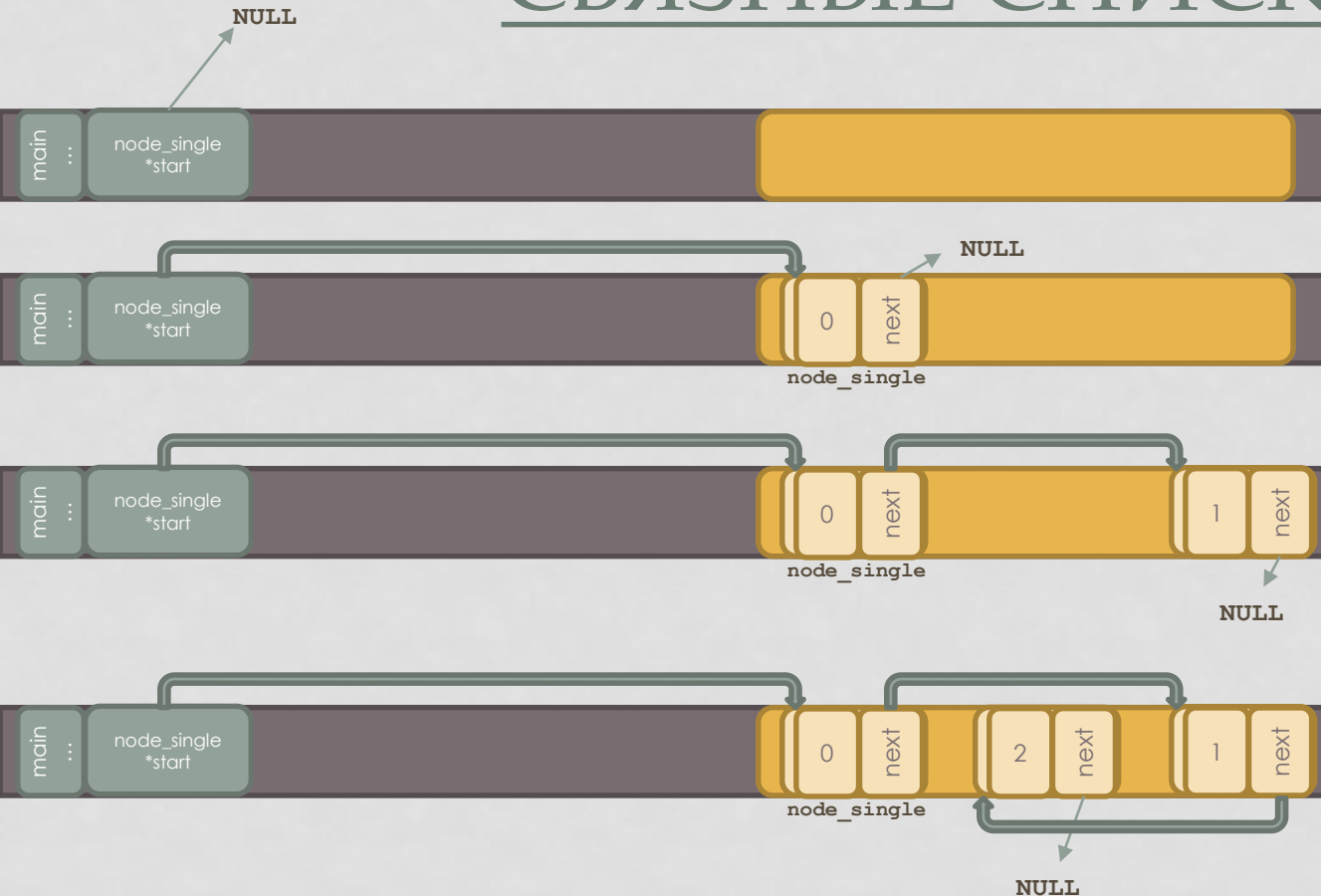
```
int main() {  
    node_single* start = NULL;  
    start = new node_single;  
    start->data = 0;  
    start->next = NULL;  
    start->next = new node_single;  
    start->next->data = 1;  
    start->next->next = NULL;  
    start->next->next = new node_single;  
    start->next->next->data = 2;  
    start->next->next->next = NULL;  
    print_list(start);  
    delete start->next->next;  
    delete start->next;  
    delete start;  
    return 0;  
}
```

# СВЯЗНЫЕ СПИСКИ



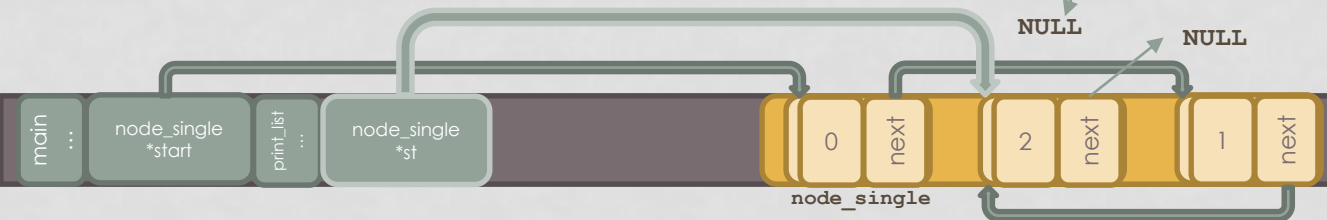
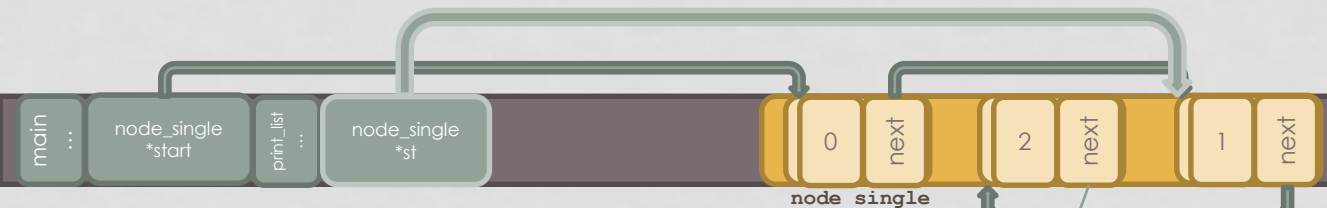
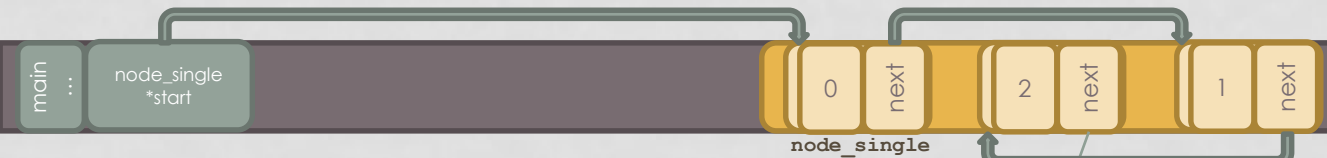
# СВЯЗНЫЕ СПИСКИ

```
struct node_single {  
    int data;  
    node_single * next;  
};
```



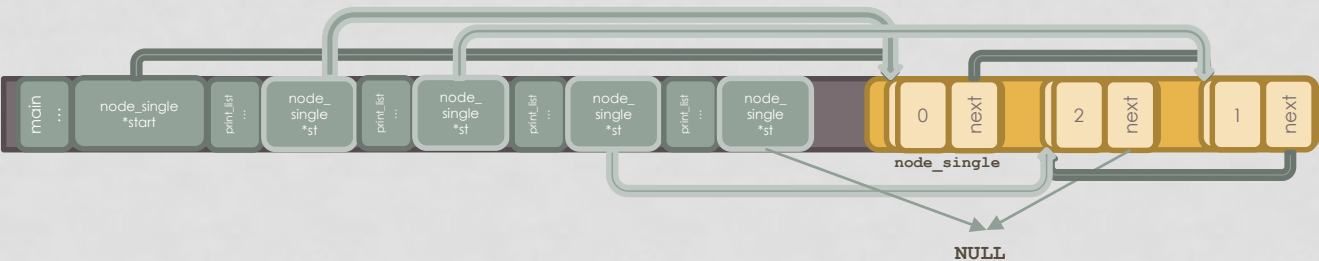
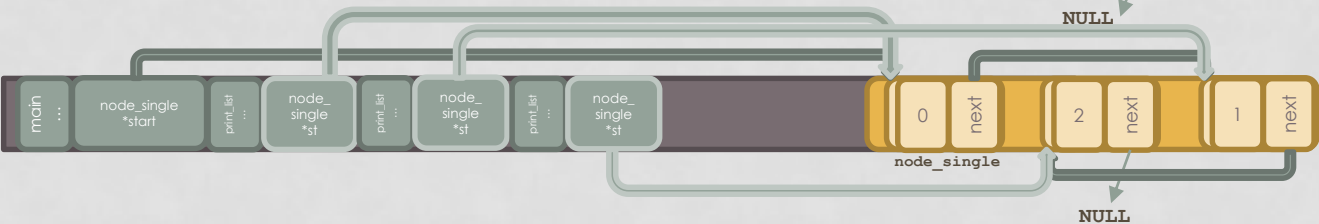
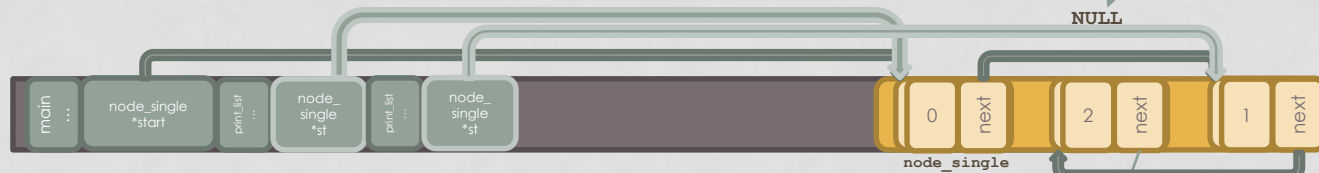
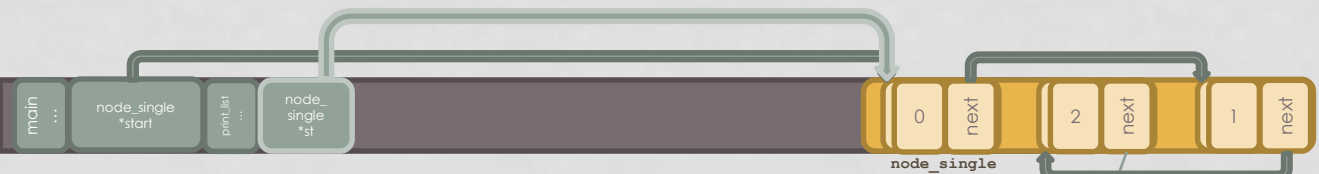
```
int main() {  
    node_single* start = NULL;  
    start = new node_single;  
    start->data = 0;  
    start->next = NULL;  
    start->next = new node_single;  
    start->next->data = 1;  
    start->next->next = NULL;  
    start->next->next = new node_single;  
    start->next->next->data = 2;  
    start->next->next->next = NULL;  
    print_list(start);  
    delete start->next->next;  
    delete start->next;  
    delete start;  
    return 0;  
}
```

# СВЯЗНЫЕ СПИСКИ



```
void print_list(node_single* st)
{
    while (st) {
        cout << st->data << " ";
        st = st->next;
    }
}
```

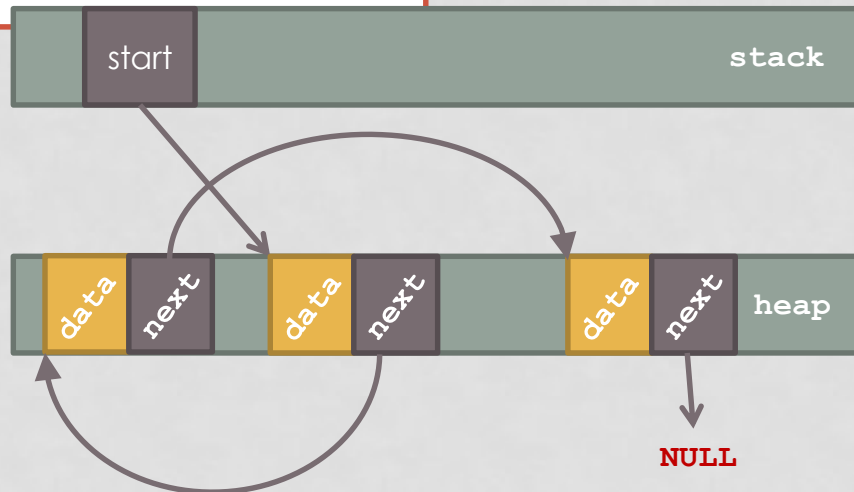
# СВЯЗНЫЕ СПИСКИ



```
void print_list(node_single* st)
{
    if (!st) return;
    cout << st->data << " ";
    print_list(st->next);
}
```

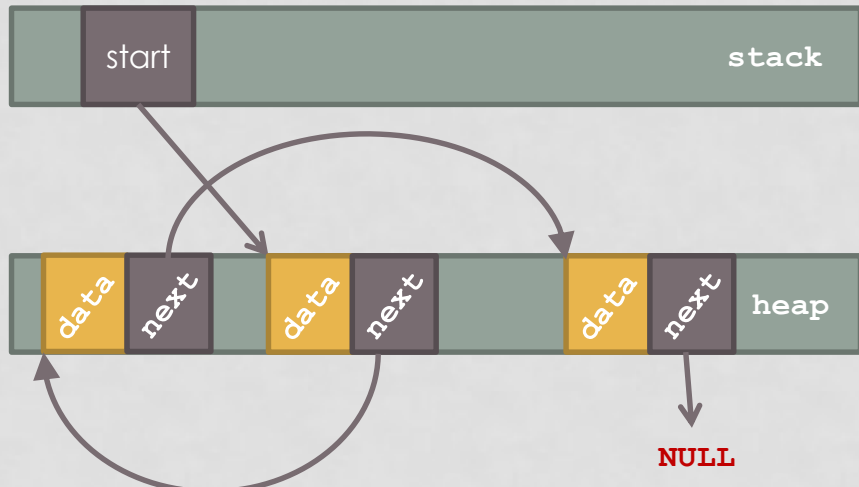
# СВЯЗНЫЕ СПИСКИ

```
void insert_start(node_single* start, int data)
{
    node_single* temp = new node_single;
    temp->data = data;
    temp->next = start;
    start = temp;
}
```

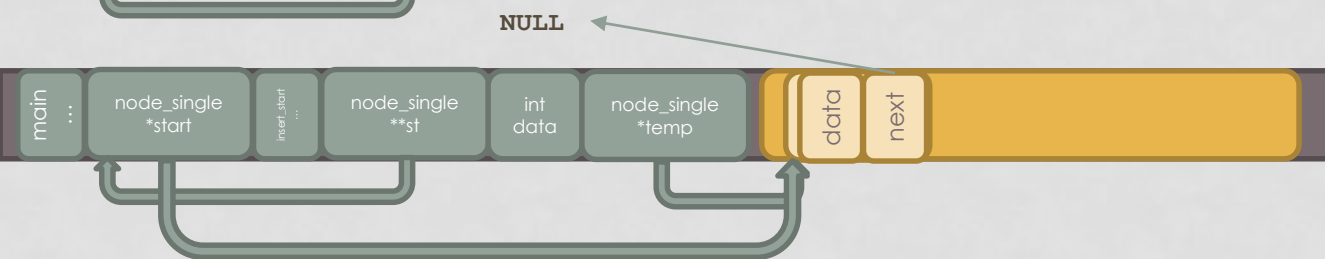
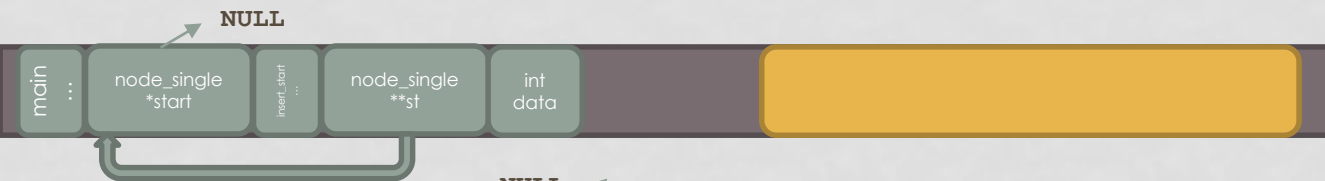
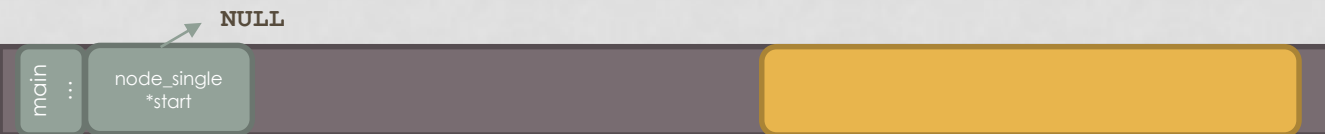


# СВЯЗНЫЕ СПИСКИ

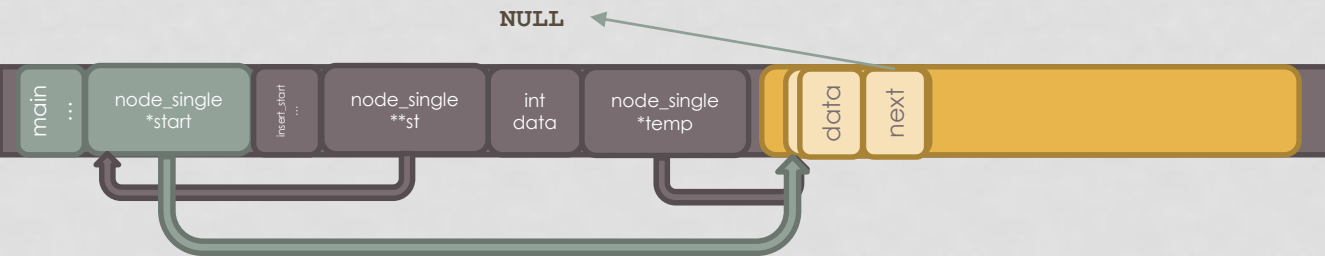
```
void insert_start(node_single** start, int data)
{
    node_single* temp = new node_single;
    temp->data = data;
    temp->next = *start;
    *start = temp;
}
```



# СВЯЗНЫЕ СПИСКИ



```
void insert_start(node_single** st, int data)
{
    node_single* temp = new node_single;
    temp->data = data;
    temp->next = *st;
    *st = temp;
}
```

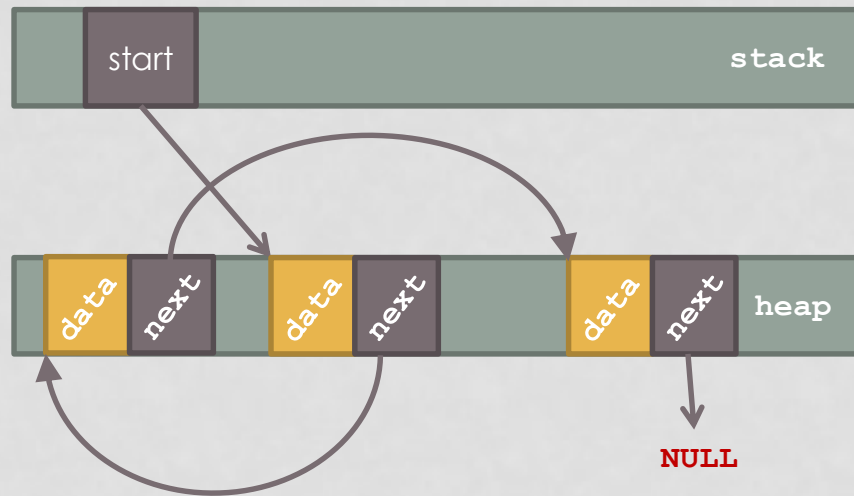




# СВЯЗНЫЕ СПИСКИ

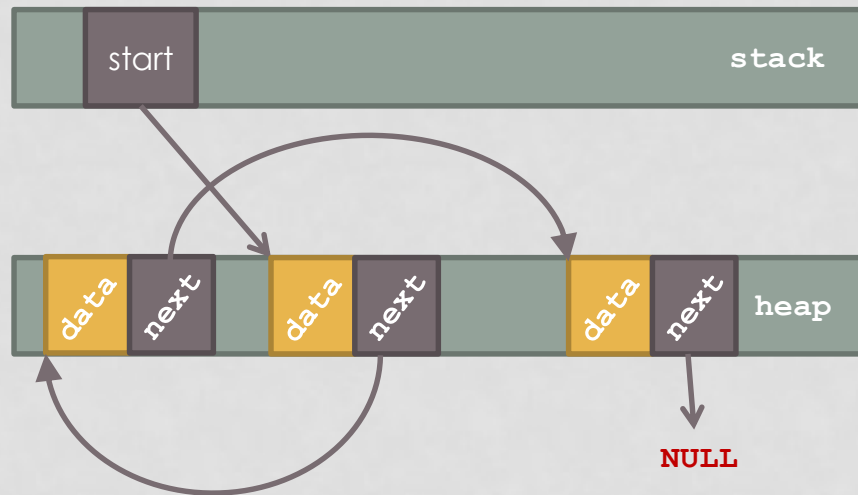
```
void very_clean(node_single** start)
{
    if (!start) return;
    very_clean(&((*start)->next));
    delete *start;
    *start = NULL;
}
```

```
void clean(node_single* start)
{
    if (!start) return;
    clean(start->next);
    delete start;
}
```




# СВЯЗНЫЕ СПИСКИ


```
int main()
{
    node_single* start = NULL;
    insert_start(&start, 1);
    insert_start(&start, 2);
    insert_start(&start, 3);
    insert_start(&start, 4);
    print_list(start);
    very_clean(&start);
    return 0;
}
```





# SUBFORWARDLIST

```
struct subforwardlist {  
    int data;  
    subforwardlist* next;  
};
```


```
bool init(subforwardlist  sfl);           //инициализация пустого недосписка
```


```
bool push_back(subforwardlist  sfl, int d);       //добавление элемента в конец недосписка
```


```
int pop_back(subforwardlist  sfl);       //удаление элемента с конца недосписка
```


```
bool push_forward(subforwardlist  sfl, int d);   //добавление элемента в начало недосписка
```

```
int pop_forward(subforwardlist  sfl);   //удаление элемента из начала недосписка
```

```
bool push_where(subforwardlist  sfl, unsigned int where, int d); //добавление элемента с поряд-  
                                                                    //ковым номером where
```

```
bool erase_where( sfl, unsigned int where);   //удаление элемента с порядковым номером where
```

```
unsigned int size(subforwardlist  sfl);       //определить размер недосписка
```

```
void clear(subforwardlist  sfl);       //очистить содержимое недосписка
```

# ТЕРМИНОЛОГИЯ

Структуры данных  
и контейнеры

Стек  
Очередь

Массивы

статические

динамические

Списки

ОДНОСВЯЗНЫЕ

ДВУСВЯЗНЫЕ

Деревья

несбалансированные

АВЛ

красно-черные

Хэш-таблицы

последовательные

ВЫ ЗДЕСЬ

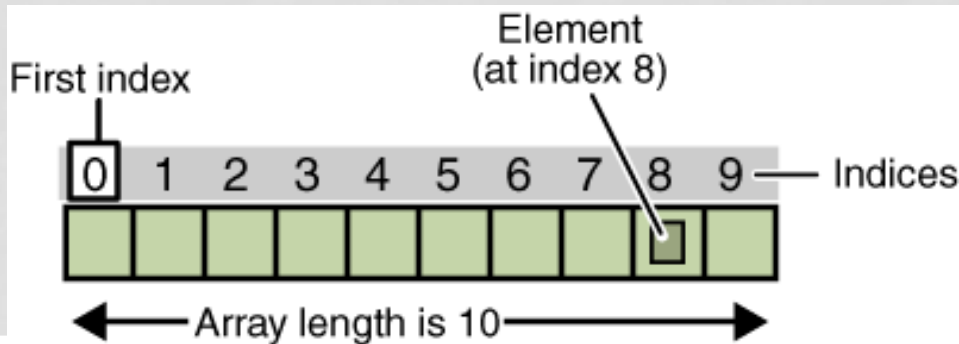
ассоциативные

неупорядоченные  
ассоциативные



# МАССИВЫ

- Все элементы лежат рядом
- занимает ровно  $N * \text{sizeof}(a[0])$  байтов
- доступ к произвольному элементу  $O(1)$
- изменение размера и добавление элемента  $O(N)$



# СВЯЗНЫЕ СПИСКИ

- элементы лежат в разных местах
- занимает дополнительную память на указатели
- доступ к произвольному элементу  $O(N)$
- можно добавлять и удалять элементы в начале списка или по указателю за  $O(1)$

