



Теория алгоритмов

Лекция 2

Способы организации данных

The screenshot displays a C++ code editor with the following code:

```
#include "gates.h"
//methods for h
void gate::it_nib
for(size_t
ins_temp[
]
void gate::it_pl
for(size_t
outs[i]
}
Inverter
gate_not(gate_
name = name
ins.reserve
outs.reserve(1);
//ins_temp.resize(ins.capacity());
//outs_temp.resize(outs.capacity());
operate() {
outs_temp[0] = ! ins_temp[0];
return false;
if(outs.capacity() != 1)
return false;
ins_temp.resize(ins.capacity());
```

The diagram shows a binary tree structure with nodes labeled 1, 8, 13, 17, 11, 15, 25, 6, 22, 27, and NIL. Arrows indicate the flow of data or operations between these nodes. The code editor also shows a Solution Explorer on the left and an Output window at the bottom.



Общая классификация структур данных



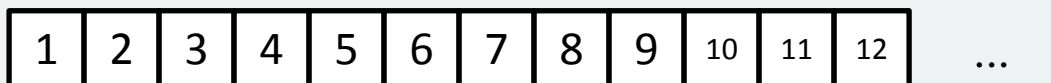


Абсолютная адресация памяти

Этап 2. Запуск программы.

Начало данных

Начало кода



Занято

Наша программа

Занято

Характеристики абсолютной адресации данных

Удобства абсолютной адресации памяти:

1. Простота реализации компилятора

Неудобства абсолютной адресации памяти:

1. Зависимость от объёма используемой памяти
2. Зависимость от количества запущенных программ
3. Отсутствие надёжности от запуска к запуску
4. Непереносимость исполняемого кода



Сегментная адресация памяти (2)



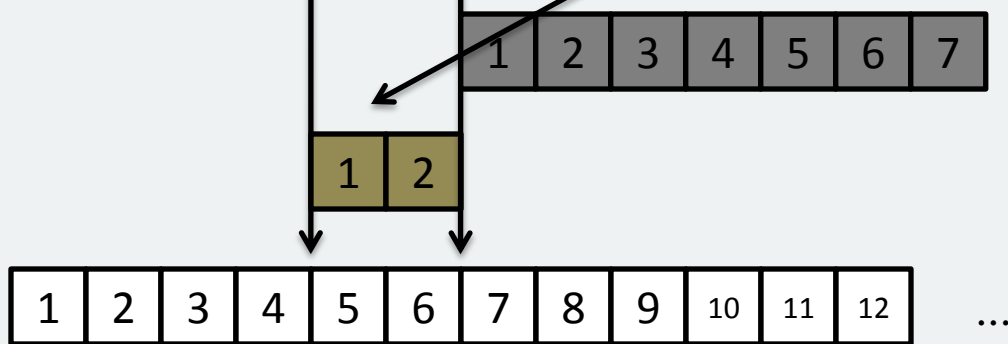
Занято операционной системой или программой

Свободно

```
int main() {
    int x = 2;
    return 0;
}
```

Начало сегмента данных

Начало сегмента кода



Характеристики сегментной адресации данных

Удобства сегментной адресации памяти:

- 1. Независимость** от объёма используемой памяти
- 2. Независимость** от количества запущенных программ
- 3. Присутствие** надёжности от запуска к запуску
- 4. Переносимость** исполняемого кода (в пределах платформы)

Неудобства сегментной адресации памяти:

1. Сложность реализации сегментирования данных – реализуется на уровне ОС



Списки

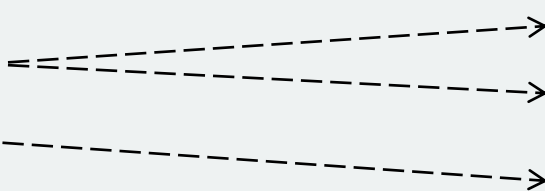
Классификация

По количеству
связей элементов

- • односвязные
- • двусвязные
- • многосвязные

По способу доступа

- • стеки (stack)
- • очереди (queue)
- • деки (deque)



Указатели

Данные

Указатели

Данные

Указатели

Данные

```

struct ListItem {
    ListItem *prev;
    ListItem *next;

    int          x;
    double     y;
    char       z[10];
};
    
```

Сравнение различных составных типов данных

	Время добавления/удаления элементов	Время доступа к элементу	Занимаемый объём памяти
Односвязный список	время выделения памяти	доступ к элементу списка	данные + указатель
Двусвязный список	время выделения памяти	доступ к элементу списка	данные + 2 указателя
Массив	выделение + копирование + удаление	минимально, «мгновенно»	только данные



Библиотека STL : контейнер `std::vector`

- | | | |
|--|---|----------------------------------|
| <code>#include <vector></code> | ← | Подключаем заголовочный файл |
| <code>using namespace std;</code> | ← | Подключаем пространство имён |
| <code>vector <int> mas;</code> | ← | Объявляем вектор целых чисел |
| <code>mas.push_back(10);</code>
<code>mas.push_back(-4);</code> | ← | Добавляем в вектор числа 10 и -4 |
| <code>cout << mas[1];</code> | ← | Обращаемся ко второму элементу |
| <code>mas.clear();</code> | ← | Очищаем массив |



Библиотека STL : контейнер `std::list`

```
#include <list>
```

← Подключаем заголовочный файл

```
using namespace std;
```

← Подключаем пространство имён

```
list <int> lst;
```

← Объявляем список целых чисел

```
lst.push_back(10);
```

```
lst.push_back(-4);
```

← Добавляем в список числа 10 и -4

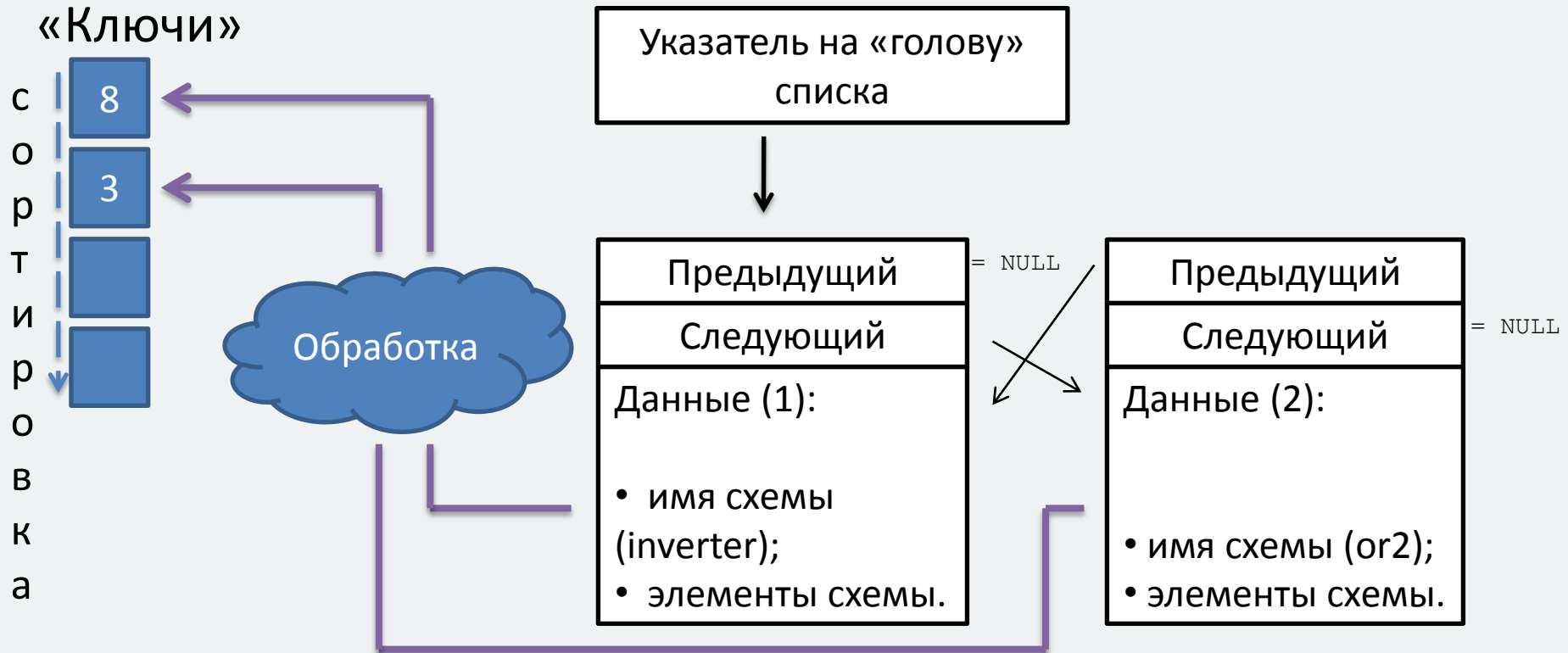
```
cout << lst.begin() + 1;
```

← Обращаемся ко второму элементу

```
lst.clear();
```

← Очищаем список

Базы данных в САПР: делаем ещё лучше





Библиотека STL : контейнер `std::map`

```
#include <map>
```

← Подключаем заголовочный файл

```
using namespace std;
```

← Подключаем пространство имён

```
map<int, float> data;
```

← Объявляем пару int, float

```
data[4] = 1;
```

```
data[2] = -3.1415;
```

```
data[-8] = 2.71828;
```

← Заносим дробные числа

Обращаемся к ключу первого элемента

```
std::cout << (*data.begin()).first << std::endl;
```

Хэш-таблицы, словари, ассоциативные массивы

Назначение: ускорение поиска данных и доступа к данным в списочных структурах

Преимущество по сравнению со списками: высокая скорость доступа

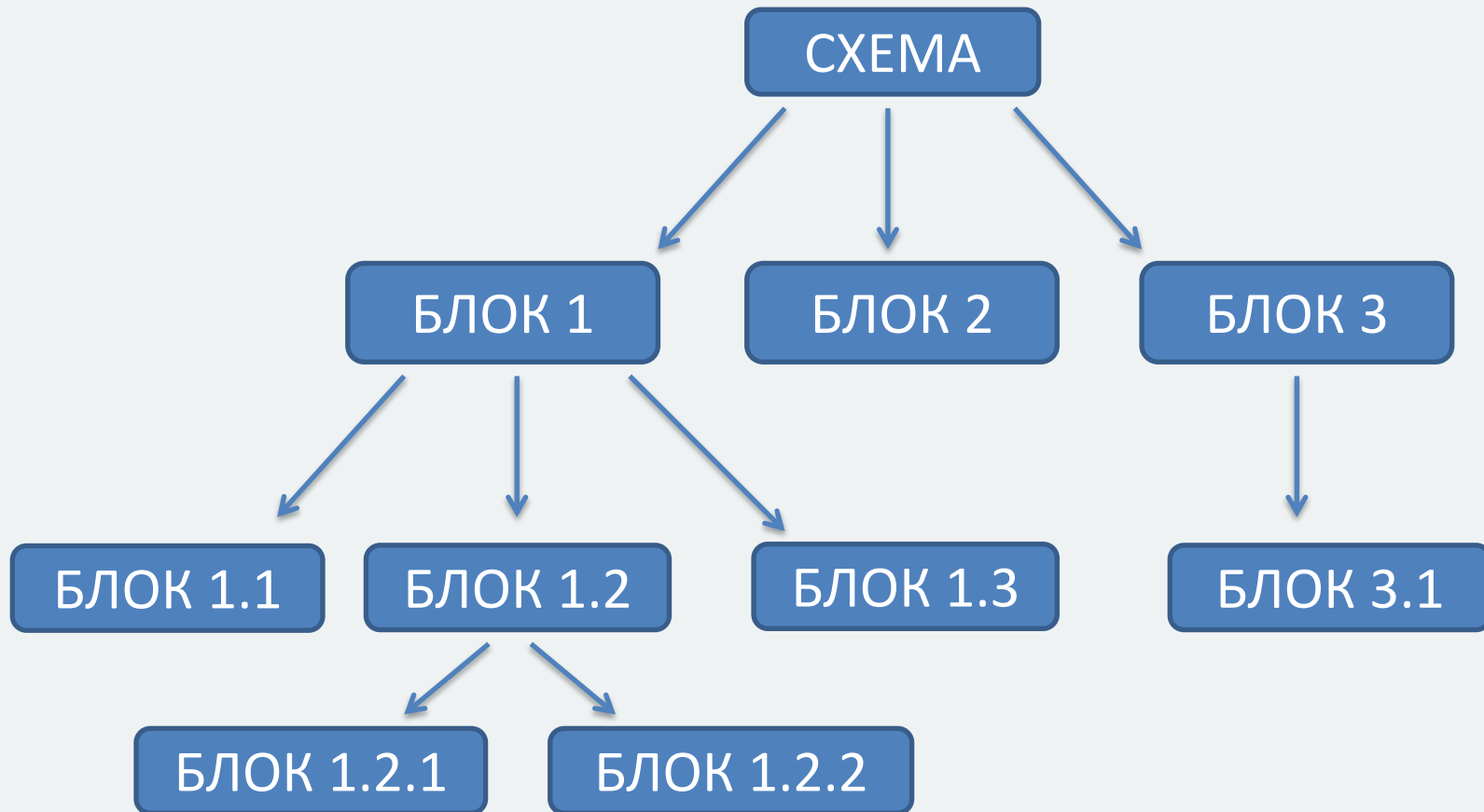
Существенный недостаток

хэш-таблиц:

использование массива для индексации:

- очень медленно при добавлении
- очень медленно при удалении
- очень быстро при обращении

Деревья как способ представления иерархии проекта





Бинарные (двоичные) деревья

