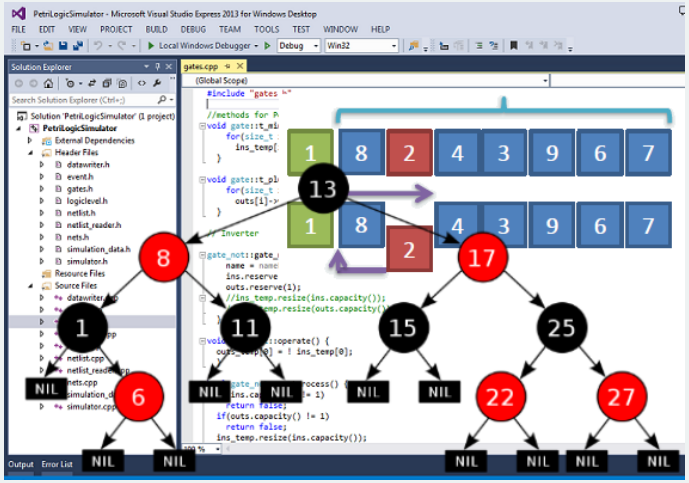
МИЭТ
Национальный исследовательский университет «МИЭТ»
Кафедра ПКИМС

Теория алгоритмов



Лекция 2

Способы организации данных

Общая классификация структур данных

Характеристики абсолютной адресации данных

Удобства абсолютной адресации памяти:

1. Простота реализации компилятора

Неудобства абсолютной адресации памяти:

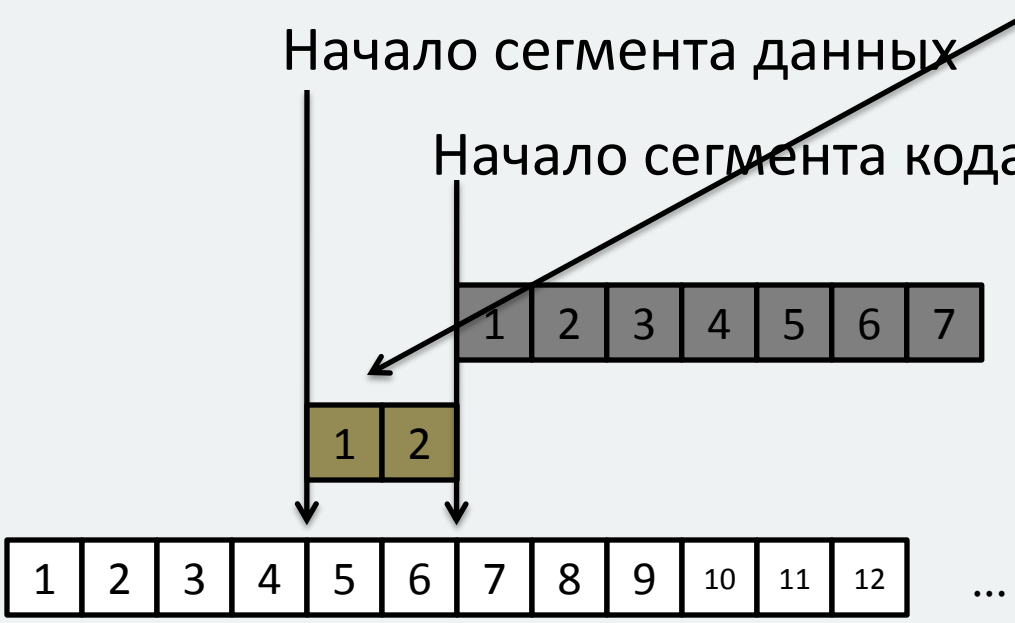
1. Зависимость от объёма используемой памяти
2. Зависимость от количества запущенных программ
3. Отсутствие надёжности от запуска к запуску
4. Непереносимость исполняемого кода

Сегментная адресация памяти (2)

Занято операционной системой или программой

Свободно

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



Характеристики сегментной адресации данных

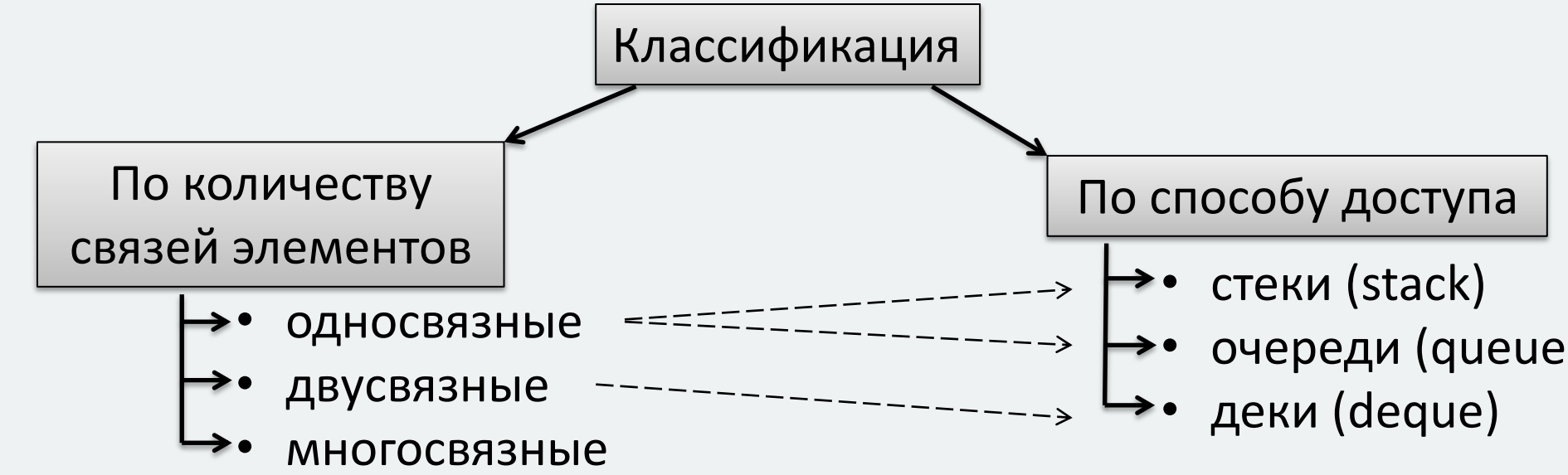
Удобства сегментной адресации памяти:

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:

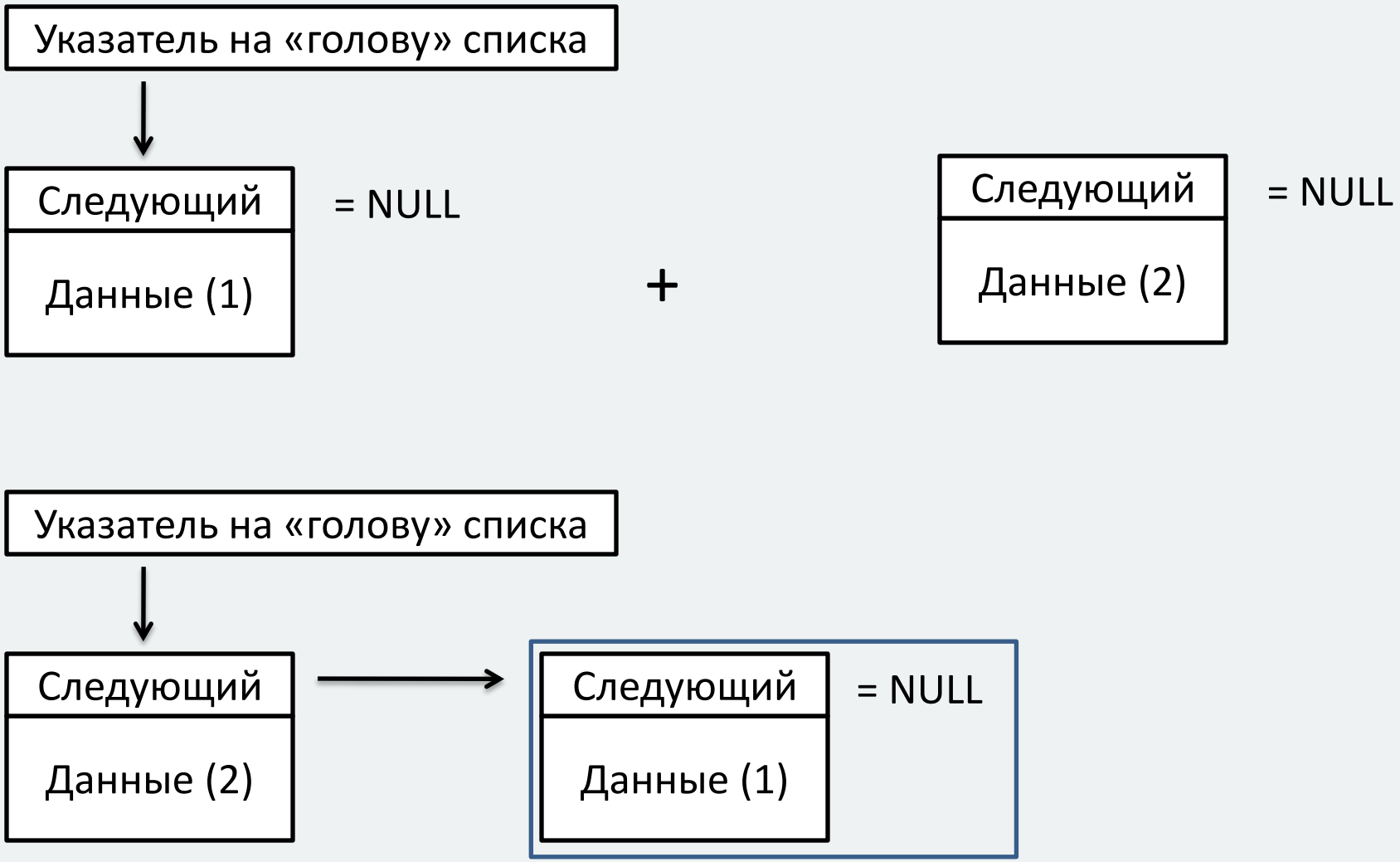
1. Сложность реализации сегментирования данных – реализуется на уровне ОС

Списки

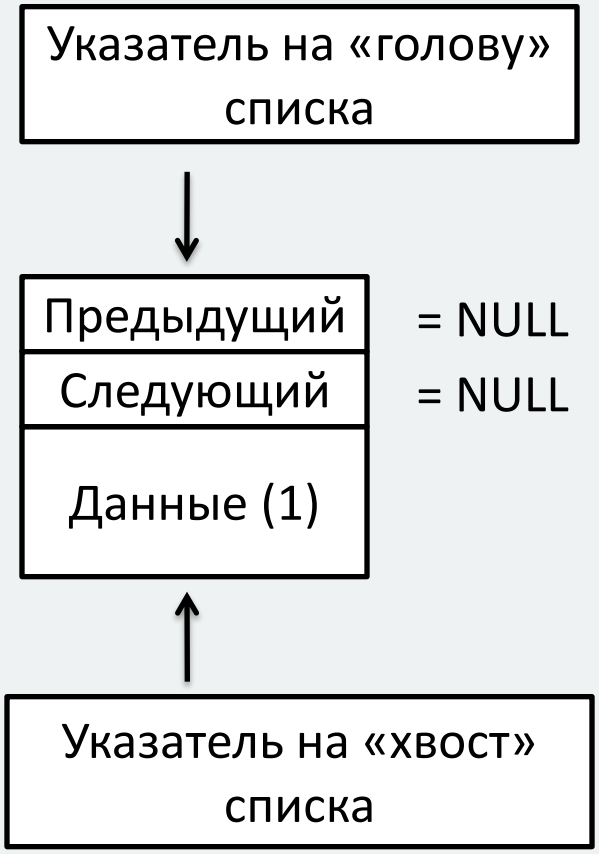


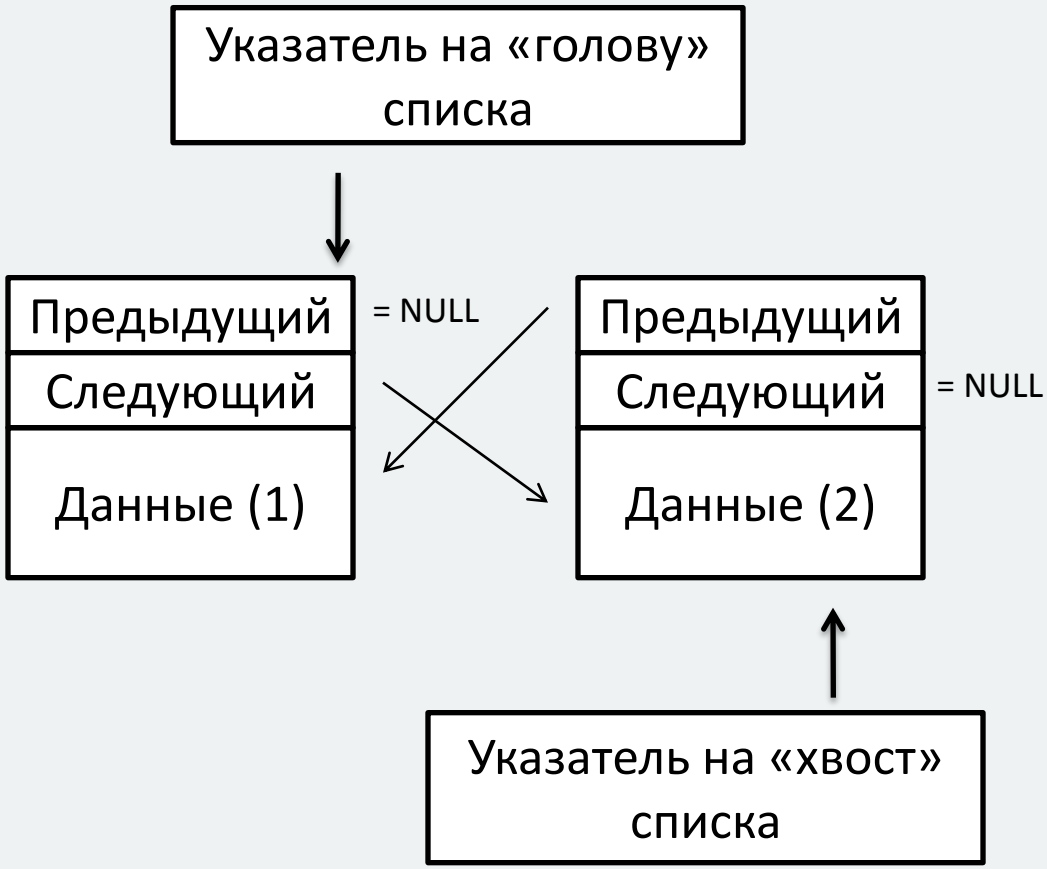

```
struct ListItem {
   ListItem *prev;
   ListItem *next;

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

Односвязные списки: стеки и очереди



Двусвязные списки: деки





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

```
#include <vector>
```
← Подключаем заголовочный файл

```
using namespace std;
```
← Подключаем пространство имён

```
vector <int>  mas;
```
← Объявляем вектор целых чисел

```
mas.push_back(10);
mas.push_back(-4);
```
← Добавляем в вектор числа 10 и -4

```
cout << mas[1];
```
← Обращаемся ко второму элементу

```
mas.clear();
```
← 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

Библиотека STL : контейнер std::map

```
#include <map>
```
← Подключаем заголовочный файл

```
using namespace std;
```
← Подключаем пространство имён

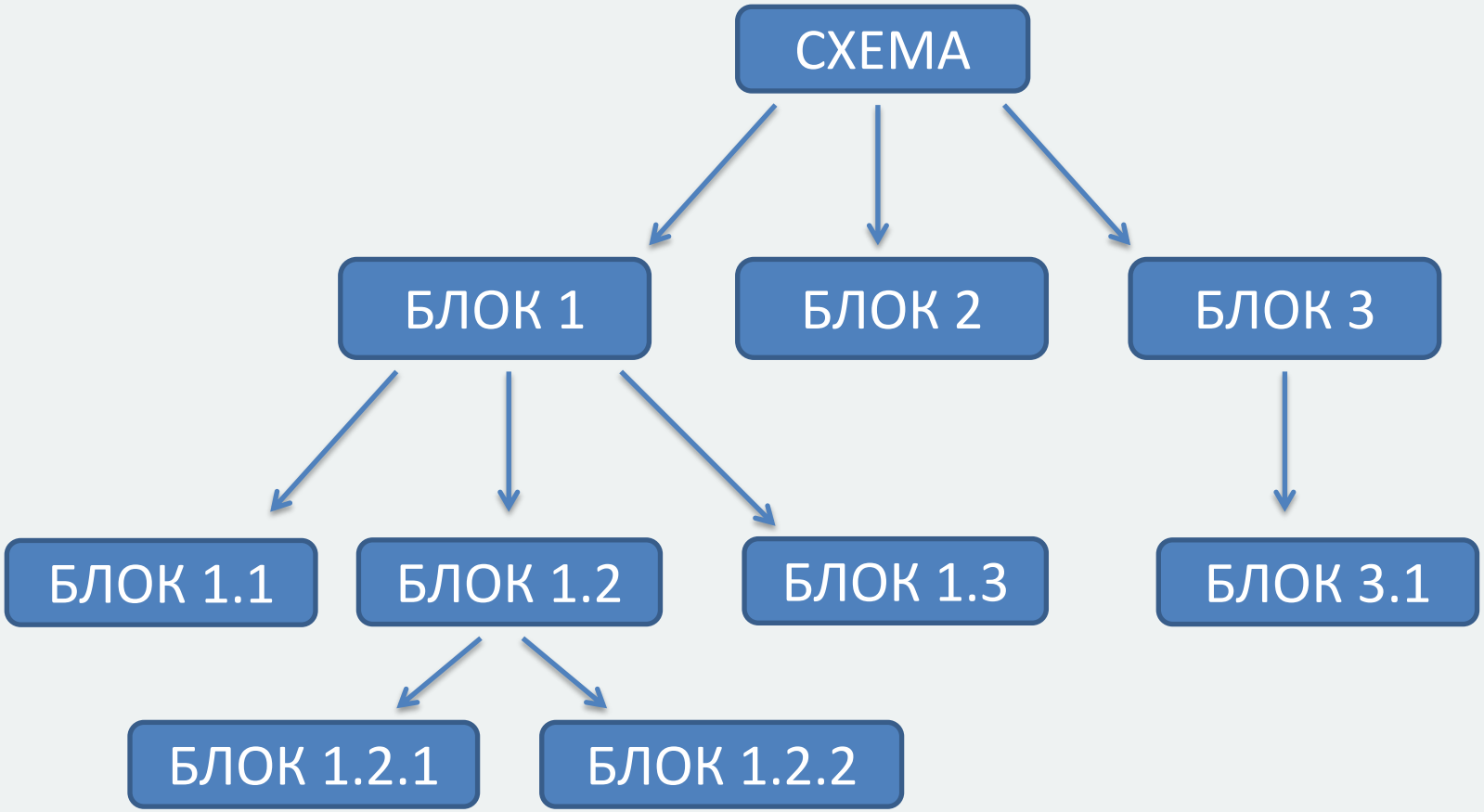
```
map<int, float>  data;
```
← Объявляем пару int, float

```
data[4] = 1;
data[2] = -3.1415;
data[-8]= 2.71828;
```
← Заносим дробные числа

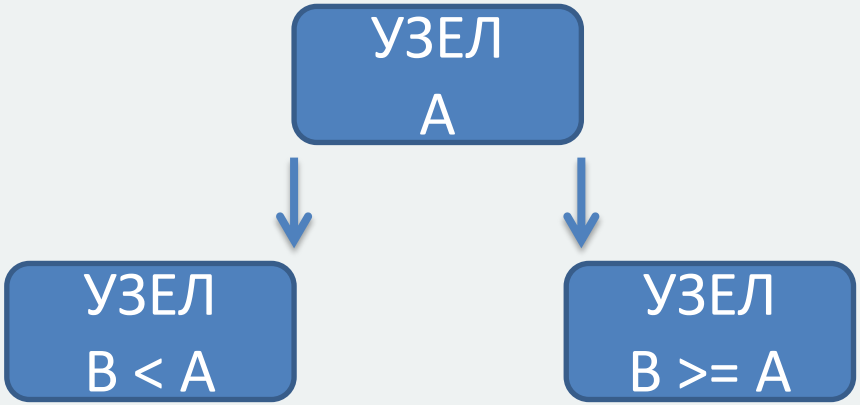
Обращаемся к ключу первого элемента

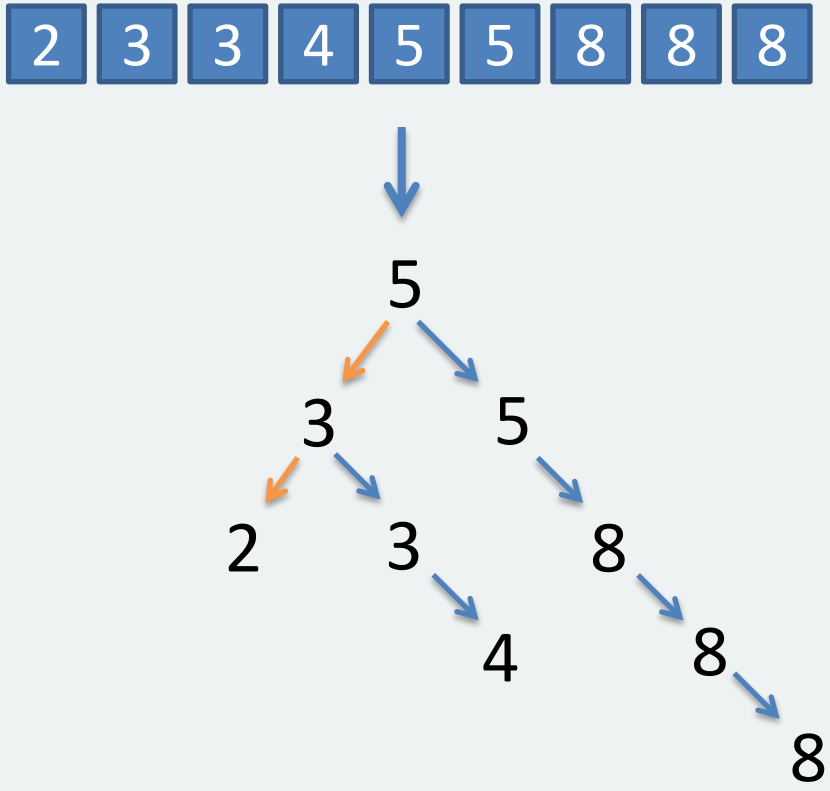
```
std::cout << (*data.begin()).first << std::endl;
```

Деревья как способ представления иерархии проекта



Бинарные (двоичные) деревья





Качество балансировки бинарного дерева

Основная задача бинарных деревьев поиска – ускорение поиска

Хорошо сбалансированное дерево