**Вопрос 1**

```
// На каких STL контейнерах может быть построен stack? Почему? (Какие требования он накладывает на интерфейс контейнера)?
Должны быть псевдонимы: value_type, reference, const_reference, size_type
Конструкторы: дефолтный, копирования
Оператор присваивания
Методы: empty, size, back, pop_back, push_back, emplace_back, swap
Операторы сравнения: ==, !=, <, <=, >, >=
Такими контейнерами могут быть: vector, deque, list
```

**Вопрос 2**

```
// На каких STL контейнерах может быть построен queue? Почему? (Какие требования он накладывает на интерфейс контейнера)?
Должны быть псевдонимы: value_type, reference, const_reference, size_type
Конструкторы: дефолтный, копирования
Оператор присваивания
Методы: empty, size, back, front, pop_front, push_front, emplace_back, swap
Операторы сравнения: ==, !=, <, <=, >, >=
Такими контейнерами могут быть: vector, deque, list
```

**Вопрос 3**

```
std::map<int, std::string> m = { /* заполняется какими-то парами */};

// Какие-то ключ и значение вводятся пользователем:
int key;
std::string value;
std::cin >> key;
std::cin >> value;

if (m.count(key) == 1) m[key] = value;// Если значение уже было, то меняем его
// Иначе бездействуем (т.е. новую пару <ключ, значение> добавлять не нужно)
// Можно ли это как-то оптимизировать? МОЖНО!
auto it = m.find(key);
if (it != m.end()) it->second = value;
// так будет только 1 поиск по дереву в find, а не как в исходной реализации, где
// 2 поиска по дереву: в count и в []
```

**Вопрос 4**

```
using numbers_t = std::map<size_t, std::string>;
numbers_t numbers = { {1, "first"}, {2, "second"}, {3, "third"} };
for (numbers_t::iterator it = numbers.begin(); it != numbers.end(); ++it) {
     const std::pair<size_t, std::string>& p = *it;
     std::cout << p.first << ": " << p.second << '\n';
}
// Цель была проитерироваться по всем элементам контейнера, не копируя их – удалось?
НЕ УДАЛОСЬ: т.е. тут ссылка на пару с неконстантным ключом, а у map ключ пары обязательно должен быть константный, иначе пользователь сможет его снаружи изменить, а это изменило бы отсортированность красно-черного дерева. Поэтому компилятор тут создаёт новую rvalue пару от которой берёт rvalue ссылку… - а это лишняя копия! Исправим:
const std::pair<const size_t, std::string>& p = *it;
```

**Вопрос 5**

```
// Упростите синтаксис итерирования по numbers (из вопроса 4), чтобы получилось:
std::map<size_t, std::string> numbers = {{1, "first"}, {2, "second"}, {3, "third"}};
for (const auto& [key, value]: numbers)
     std::cout << key << ": " << value << '\n';
```

## Вопрос 6

// Нужно написать собственный класс компаратор, (по аналогии с аналогичным компаратором из стандартной библиотеки) для того, чтобы элементы распечатались по убыванию ключей:

```
std::map<size_t, std::string, Greater<size_t>> numbers;
numbers = { {1, "first"}, {2, "second"}, {3, "third"} };
for (const auto& [key, value]: numbers)
	std::cout << key << ": " << value << '\n';
```

```
3: third
2: second
1: first
```

// Вот такой ответ засчитывается:

```
template<typename T>
struct Greater {
	bool operator()(const T& a, const T& b) const {
		return a > b;
	}
};
```

// Но если мы будем проводить аналогию с компаратором из стандартной библиотеки, то обнаружим, что начиная с C++14 появились значение по умолчанию и специализация:

```
template<typename T = void>
struct Greater {
	bool operator()(const T& a, const T& b) const {
		return a > b;
	}
};
```

```
template<>
struct Greater<void> {
	template<typename T1, typename T2>
	bool operator()(const T1& a, const T2& b) const {
		return a > b;
	}
};
```

// Это позволит сделать так: std::map<size_t, std::string, Greater<>> numbers;