

# ЛАБОРАТОРНАЯ РАБОТА №7 ПО ПРЕДМЕТУ «ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ И ПРОГРАММИРОВАНИЕ». «ВВЕДЕНИЕ В СТАНДАРТНУЮ БИБЛИОТЕКУ ШАБЛОНОВ. АДАПТЕРЫ КОНТЕЙНЕРОВ»

Стандартная библиотека *STL* предоставляет различные типобезопасные контейнеры для хранения коллекций связанных объектов. При объявлении переменной контейнера указывается тип элементов, которые будет содержать контейнер. Например, в следующем операторе объявляется контейнер типа *vector*, содержащий 5 значений типа *int*:

```
vector<int> rating(5);
```

Контейнеры могут создаваться с использованием списка инициализаторов. Например, в следующем операторе объявляется и одновременно инициализируется контейнер типа *vector*:

```
vector<double> payments({ 45.99, 39.23, 19.95, 89.01 });
```

Контейнеры содержат методы для добавления и удаления элементов контейнера и выполнения других операций над элементами контейнера.

Доступ к элементам контейнера может быть реализован с помощью итераторов. Итераторы для всех контейнеров *STL* имеют общий интерфейс, но каждый контейнер определяет собственные специализированные итераторы.

**Контейнеры можно разделить на три категории:** последовательные контейнеры, ассоциативные контейнеры и контейнеры-адаптеры.

## КОНТЕЙНЕРЫ-АДАПТЕРЫ

Существуют структуры данных, которые часто используются в решении некоторого типа задач. Эти структуры данных имеют ограниченный набор операций с элементами. Любой, подходящий для этих целей, последовательный контейнер можно использовать для реализации таких структур. Чтобы реализовать работу с такими структурами как стек или очередь, в C++ используются адаптеры контейнеров. **Контейнер-адаптер — это разновидность последовательного или ассоциативного контейнера, который ограничивает интерфейс для простоты и ясности. Контейнеры-адаптеры не поддерживают итераторы.**

Понятие «адаптер» в объектно-ориентированном проектировании и программировании описывает сущность, которая не реализует собственную функциональность, а вместо этого предоставляет альтернативный интерфейс к функциональности другой сущности.

Библиотека *STL* содержит три контейнера-адаптера:

**1. Контейнер *queue*** соответствует принципу обработки элементов *FIFO* (*first in – first out*, первым поступил — первым обслужен). Первый элемент, помещенный в очередь, должен первым извлекается из очереди. Очередь в проектировании и программировании используется, когда нужно

совершить какие-то действия в порядке их поступления, выполнив их последовательно. Примером может служить организация событий в операционной системе *Windows*.

**2. Контейнер *priority\_queue*** соответствует принципу обработки элементов *FIFO* (*first in – first out*, первым поступил — первым обслужен), но первым в очереди всегда оказывается элемент с наибольшим значением определенной характеристики (*priority*, приоритетом). В качестве примера очереди с приоритетом можно рассмотреть список задач работника. Когда работник заканчивает одну задачу, он переходит к следующей самой приоритетной. Начальник добавляет задачи в список, указывая их приоритет.

**3. Контейнер *stack*** соответствует принципу обработки элементов *LIFO* (*last in – first out*, последним поступил — первым обслужен). Последний элемент, помещенный в стек, становится первым извлекаемым из стека элементом.

**Поскольку контейнеры-адаптеры не поддерживают итераторы, их нельзя использовать в алгоритмах *STL*.**

**Адаптер (*adaptor*)** — это фундаментальная концепция библиотеки. Существуют адаптеры контейнера, итератора и функции. По существу, адаптер — это механизм, заставляющий нечто одно действовать как нечто другое. **Адаптер контейнера получает контейнер существующего типа и заставляет его действовать как другой.** Например, адаптер *stack* получает любой из последовательных контейнеров (*array* и *forward list*) и заставляет его работать подобно стеку. Функции и типы данных, общие для всех адаптеров контейнеров, перечислены в таблице №1.

Таблица №1 – Функции и типы, общие для всех контейнеров адаптеров

<i>size_type</i>	Беззнаковый целочисленный тип, который может представлять количество элементов в контейнере
<i>value_type</i>	Тип, представляющий тип объекта, хранящегося как элемент в контейнере
<i>container_type</i>	Тип контейнера, на базе которого реализован контейнер адаптер
<i>A a;</i>	Создает новый пустой адаптер с именем <i>a</i>
<i>A a (c);</i>	Создает новый адаптер с именем <i>a</i> , содержащий копию контейнера <i>c</i>
<i>операторы сравнения</i>	Каждый адаптер поддерживает все операторы сравнения: <i>==</i> , <i>!=</i> , <i>&lt;</i> , <i>&lt;=</i> , <i>&gt;</i> , <i>&gt;=</i> . Эти операторы возвращают результат сравнения внутренних контейнеров
<i>a.empty()</i>	Возвращает значение <i>true</i> , если адаптер <i>a</i> пуст, и значение <i>false</i> в противном случае
<i>a.size()</i>	Возвращает количество элементов в адаптере <i>a</i>
<i>swap (a, b)</i> <i>a.swap (b)</i>	Меняет содержимое контейнеров <i>a</i> и <i>b</i> ; у них должен быть одинаковый тип, включая тип контейнера, на основании которого они реализованы

Каждый контейнер адаптер определяет два конструктора: стандартный конструктор, создающий пустой объект, и конструктор, получающий контейнер и инициализирующий адаптер, копируя полученный контейнер. Предположим, например, что *deq* — это двусторонняя очередь *deque<int>*. Ее можно использовать для инициализации нового стека следующим образом:

```
deque<int> deq;
```

```
stack<int> stk(deq); //копирует элементы из deq в stk
```

По умолчанию контейнеры-адаптеры, *stack* и *queue*, реализованы на основании контейнера *deque*. Контейнер-адаптер *priority\_queue* может быть реализован как на базе контейнера *vector* так, и на базе *deque*.

Заданный по умолчанию тип контейнера можно переопределить, указав последовательный контейнер как второй аргумент при создании адаптера:

```
//пустой стек, реализованный поверх вектора
stack<string, vector<string>> strStack1;
vector<string> strVector2;
strVector2.push_back("firstString");
strVector2.push_back("secondString");
strVector2.push_back("thirdString");
//strStack3 реализован поверх вектора, первоначально содержит копию
strVector2
stack<string, vector<string>> strStack3(strVector2);
```

Существуют некоторые ограничения на применение контейнеров с определенными адаптерами.

Всем адаптерам нужна возможность добавлять и удалять элементы. **В результате адаптеры не могут быть основаны на массиве. Точно также нельзя использовать контейнер *forward list*, поскольку все адаптеры требуют функций добавления, удаления и обращения к последнему элементу контейнера.**

Адаптер *stack* требует только функций *push()*, *pop()*, *top()*, поэтому для стека можно использовать контейнер любого из остальных типов.

Адаптеру *queue* требуются функции *back()*, *front()*, *push()*, *pop()*, поэтому он может быть создан на основании контейнеров *list* и *deque*, но не *vector*.

Адаптеру *priority\_queue* требуются функции *top()*, *push()*, *pop()*; он может быть основан на контейнерах *vector* и *deque*, но не *list*.

## АДАПТЕР STACK

Стек — это структура данных, в которой элементы добавляются и удаляются в вершине стека. Набор элементов в стеке организован по принципу *LIFO* (*last in — first out*, «последним пришёл — первым вышел»). Обычно такую структуру представляют как стопку тарелок: доступ ко второй (сверху) тарелке можно получить только после того, как будет взята первая. Стек используется при разборе (*parsing*) блоков данных, как средство моделирования рекурсии, как модель исполнения инструкций.

**В основу адаптера *stack* положен (по умолчанию) контейнер *deque*.**

В программировании, стек играет очень важную роль, если не самую главную. В иерархии вызовов методов (*call history*) всегда будет именно стек. Рекурсивные вызовы методов тоже используют стек. Отсюда и классическая проблема *stack overflow*, когда стек заполняется до отказа выполняемыми методами и происходит физическое ограничение по памяти или возможностям компьютера или операционной системы.

Тип *stack* определен в заголовке `<stack>`. Методы-элементы класса *stack* перечислены в таблице №2.

Таблица №2 – Основные функции, поддерживаемые контейнером адаптером *stack*

Метод	Выполняемое действие
<i>bool empty() const</i>	Проверяет, пуст ли стек
<i>void pop()</i>	Удаляет элемент из верхней части стека
<i>void push(const Type&amp; val)</i>	Добавляет элемент в верхнюю часть стека
<i>size_type size() const</i>	Возвращает количество элементов в стеке
<i>reference top()</i> <i>const_reference top() const</i>	Возвращает ссылку на элемент в вершине стека

Следующая программа иллюстрирует использование адаптера *stack*:

```
//Пример №1. Использование класса stack
#include <stack>
#include <iostream>
using namespace std;
int main() {
    stack<int> intStack; //пустой стек
    //заполнить стек
    for (size_t i = 0; i != 10; ++i)
        intStack.push(i); //добавление в intStack значений от 0 до 9
    while (!intStack.empty()) { //пока в intStack есть значения
        cout << intStack.top() << endl;
        intStack.pop(); //извлечь верхний элемент и повторить
    }
    return 0;
}
```

Результаты работы программы:

```
9
8
7
6
5
4
3
2
1
0
```

Сначала *intStack* объявляется как пустой стек целочисленных элементов. Затем цикл *for* добавляет десять элементов, инициализируя каждый следующим целым числом, начиная с нуля. Цикл *while* перебирает весь стек, извлекая и выводя верхний элемент, пока стек не опустеет.

Хотя стек реализован на основании контейнера, прямого доступа к функциям контейнера *deque* нет. Для стека нельзя вызвать функцию *push\_back()*, вместо нее следует использовать функцию *push()*.

**По умолчанию адаптер *stack* реализован на базе контейнера *deque*, но адаптер *stack* может быть также реализован на базе контейнеров *list* или *vector*.**

```
//Пример №2. Использование класса stack
#include <stack>
#include <iostream>
using namespace std;
int main() {
```

```

system("chcp 1251");
system("cls");
stack<int> s1, s2;
s1.push(10);
s1.push(20);
s1.push(30);
stack<int>::size_type i;
i = s1.size();
cout << "Размер стека равен " << i << "." << endl;
i = s1.top();
cout << "Элемент на вершине стека " << i << "." << endl;
s1.pop();
i = s1.size();
cout << "После извлечения элемента из стека размер равен " << i
<< "." << endl;
i = s1.top();
cout << "После извлечения элемента из стека элемент на вершине
стека " << i << "." << endl;
}

```

Результаты работы программы:

```

Размер стека равен 3.
Элемент на вершине стека 30.
После извлечения элемента из стека размер равен 2.
После извлечения элемента из стека элемент на вершине стека 20
.

```

**Каждый контейнер-адаптер определяет собственные функции, исходя из функций, предоставленных базовым контейнером. Использовать можно только функции адаптера, а функции основного контейнера использовать нельзя.**

### АДАПТЕРЫ ОЧЕРЕДЕЙ

Адаптеры *queue* и *priority\_queue* определены в заголовке `<queue>`. Список функций, поддерживаемых этими типами, приведен в таблице №3.

По умолчанию адаптер *queue* использует контейнер *deque*, а адаптер *priority\_queue* — контейнер *vector*; адаптер *queue* может использовать также контейнер *list* или *vector*, адаптер *priority\_queue* может использовать контейнер *deque*.

Таблица №3 – Основные функции адаптеров *queue* и *priority\_queue*

Метод	Выполняемое действие
<i>reference back()</i> <i>const reference back() const</i>	Возвращает ссылку на последний добавленный элемент в конце очереди. Используется для контейнера <i>queue</i>
<i>bool empty() const</i>	Проверяет, пуста ли очередь. Используется для контейнера <i>queue</i> и <i>priority_queue</i>
<i>reference front()</i> <i>const reference front() const</i>	Возвращает ссылку на первый элемент в начале очереди. Используется для контейнера <i>queue</i>
<i>void pop()</i>	Удаляет элемент из начала очереди. Удаляет самый большой элемент из <i>priority_queue</i> из начала
<i>void push(const Type&amp; val)</i>	Добавляет элемент в конец очереди. Добавляет элемент в <i>priority_queue</i> на основании приоритета элемента
<i>size_type size() const</i>	Возвращает количество элементов в очереди

<code>const_reference top() const</code>	Возвращает, но не удаляет элемент с самым высоким приоритетом. Возвращает константную ссылку на самый большой элемент в начале <i>priority_queue</i>
--	--

Библиотечный класс *queue* использует хранилище, организованное по принципу "первым пришел, первым вышел" (*first-in, first-out* — *FIFO*). Поступающие в очередь объекты помещаются в ее конец, а извлекаются из ее начала, при этом выбранный элемент из очереди удаляется.

Адаптер *priority\_queue* позволяет установить приоритет хранимых элементов. Добавляемые элементы помещаются перед элементами с более низким приоритетом. По умолчанию для определения относительных приоритетов в библиотеке используется оператор `<`.

```
//Пример №3. Использование методов front() и back() контейнера queue
#include <queue>
#include <iostream>
using namespace std;
int main() {
    system("chcp 1251");
    system("cls");
    queue<int> queueInt;
    queueInt.push(111);
    queueInt.push(101);
    queueInt.push(110);
    queueInt.push(12);
    queueInt.push(12);
    queueInt.push(13);
    int& element = queueInt.back();
    const int& otherElement = queueInt.front();
    cout << "Номер, находящийся в конце очереди:" << element << "."
<< endl;
    cout << "Номер, находящийся в начале очереди:" << otherElement <<
    "." << endl;
}
```

Результаты работы программы:

```
Номер, находящийся в конце очереди:13.
Номер, находящийся в начале очереди:111.
```

```
//Пример №4. Использование класса priority_queue
#include <queue>
#include <iostream>
using namespace std;
int main() {
    system("chcp 1251");
    system("cls");
    priority_queue<int> intQueue;
    intQueue.push(10);
    intQueue.push(30);
    intQueue.push(20);
    priority_queue<int>::size_type queueSize;
    queueSize = intQueue.size();
}
```

```

    cout << "Длина очереди с приоритетом равна " << queueSize << "."
<< endl;
    const int& ii = intQueue.top();
    cout << "В начале очереди находится элемент с номером " << ii <<
    "." << endl;
}

```

Результаты работы программы:

```

Длина очереди с приоритетом равна 3.
В начале очереди находится элемент с номером 30.

```

//Пример №5. Использование класса priority\_queue с указанием компаратора

```

#include <queue>
#include <vector>
#include <iostream>
using namespace std;
template<typename T>
void printQueue(T& queue) {
    while (!queue.empty()) {
        cout << queue.top() << " ";
        queue.pop();
    }
    cout << '\n';
}
int main() {
    priority_queue<int> queue;
    for (int n : {1, 8, 5, 6, 3, 4, 0, 9, 7, 2})
        queue.push(n);
    printQueue(queue);
    priority_queue<int, vector<int>, greater<int>> q2;
    for (int n : {1, 8, 5, 6, 3, 4, 0, 9, 7, 2})
        q2.push(n);
    printQueue(q2);
}

```

Результаты работы программы:

```

9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9

```

//Пример №6. Использование класса priority\_queue с пользовательскими типами

```

#include <iostream>
#include <queue>
#include <iomanip>
using namespace std;
struct Time {
    int h; // >= 0
    int m; // 0-59
    int s; // 0-59
};
class CompareTime {

```



```

public:
    bool operator()(Time& t1, Time& t2)
    {
        if (t1.h < t2.h) return true;
        if (t1.h == t2.h && t1.m < t2.m) return true;
        if (t1.h == t2.h && t1.m == t2.m && t1.s < t2.s) return
true;
        return false;
    }
};

```

Descending greater

```

//Третий параметр шаблонного класса должен быть классом, у которого
operator()
priority_queue<Time, vector<Time>, CompareTime> pq;
int main() {
    priority_queue<Time, vector<Time>, CompareTime> pq;
    // Array of 4 time objects:
    Time t[4] = { {3, 2, 40}, {3, 2, 26}, {5, 16, 13}, {5, 14, 20} };
    for (int i = 0; i < 4; ++i)
        pq.push(t[i]);
    while (!pq.empty()) {
        Time t2 = pq.top();
        cout << setw(3) << t2.h << " " << setw(3) << t2.m << " " <<
            setw(3) << t2.s << endl;
        pq.pop();
    }
    return 0;
}

```

Результаты работы программы:

5	16	13
5	14	20
3	2	40
3	2	26

//Пример №7. Использование класса priority\_queue с указанием способа сравнения

```

#include <queue>
#include <string>
#include <iostream>
using namespace std;
class Student {
public:
    string chName;
    int nAge;
    Student() : chName(" "), nAge(0) {}
    Student(string chNewName, int nNewAge) : chName(chNewName),
nAge(nNewAge) {}
};
//перегрузка оператора <
bool operator< (const Student& st1, const Student& st2) {
    return st1.nAge > st2.nAge;
}
//перегрузка оператора >
bool operator> (const Student& st1, const Student& st2) {
    return st1.nAge < st2.nAge;
}

```

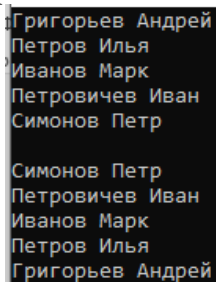


```

}
int main() {
    system("chcp 1251");
    system("cls");
    //Создание priority_queue и указание порядка элементов как <
    //Элементы очереди будут расположены в порядке увеличения
    возраста
    priority_queue<Student, vector<Student>,
less<vector<Student>::value_type>> pqStudent1;
    //Создание priority_queue и указание порядка элементов как >
    //Элементы очереди будут расположены в порядке уменьшения
    возраста
    priority_queue<Student, vector<Student>,
greater<vector<Student>::value_type>> pqStudent2;
    //добавление элементов в контейнер
    pqStudent1.push(Student("Иванов Марк", 38));
    pqStudent1.push(Student("Петров Илья", 25));
    pqStudent1.push(Student("Симонов Петр", 47));
    pqStudent1.push(Student("Григорьев Андрей", 13));
    pqStudent1.push(Student("Петровичев Иван", 44));
    //отображение элементов контейнера
    while (!pqStudent1.empty()) {
        cout << pqStudent1.top().chName << endl;
        pqStudent1.pop();
    }
    cout << endl;
    //добавление элементов в контейнер
    pqStudent2.push(Student("Иванов Марк", 38));
    pqStudent2.push(Student("Петров Илья", 25));
    pqStudent2.push(Student("Симонов Петр", 47));
    pqStudent2.push(Student("Григорьев Андрей", 13));
    pqStudent2.push(Student("Петровичев Иван", 44));
    //отображение элементов контейнера
    while (!pqStudent2.empty()) {
        cout << pqStudent2.top().chName << endl;
        pqStudent2.pop();
    }
    cout << endl;
    return 0;
}

```

Результаты работы программы:



```

Григорьев Андрей
Петров Илья
Иванов Марк
Петровичев Иван
Симонов Петр

Симонов Петр
Петровичев Иван
Иванов Марк
Петров Илья
Григорьев Андрей

```

## КОНТРОЛЬНЫЕ ВОПРОСЫ К ЛАБОРАТОРНОЙ РАБОТЕ:

1. Что такое контейнер в библиотеке *STL*?
2. Что такое итератор в библиотеке *STL*?

3. Что такое алгоритм в библиотеке *STL*?
4. Что такое итератор в *STL*? Какие виды итераторов существуют в библиотеке *STL*?
5. Что представляет собой предикат в библиотеке *STL*?
6. Что собой представляет контейнер вектор (*vector*) в *STL C++*?
7. Что собой представляет контейнер список (*list*) в *STL C++*?
8. Что собой представляет контейнер «отображение» (*map*) в *STL C++*?
9. В чем различие между контейнерами *list*, *vector*, *map*?
10. Что собой представляют ассоциативные и последовательные контейнеры? Приведите примеры. В чем между ними различие?

#### **ПОРЯДОК ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ:**

1. Изучить теоретические сведения, полученные на лекции и лабораторной работе, ознакомиться с соответствующими материалами литературных источников.

2. Ответить на контрольные вопросы лабораторной работы.

3. Разработать алгоритм программы по индивидуальному заданию.

4. Написать, отладить и проверить корректность работы созданной программы.

5. Написать электронный отчет по выполненной лабораторной работе.

**Отчет должен быть оформлен по стандарту БГУИР ([Стандарт предприятия СТП 01-2017 "Дипломные проекты \(работы\). Общие требования"](#)) и иметь следующую структуру:**

1. титульный лист
2. цель выполнения лабораторной работы
3. теоретические сведения по лабораторной работе
4. формулировка индивидуального задания
5. весь код решения индивидуального задания, разбитый на необходимые типы файлов
6. скриншоты выполнения индивидуального задания
7. выводы по лабораторной работе

**В РАМКАХ ВСЕГО КУРСА «ООП» ВСЕ ЛАБОРАТОРНЫЕ РАБОТЫ НА ЯЗЫКЕ C++ ДОЛЖНЫ ХРАНИТЬСЯ В ОДНОМ РЕШЕНИИ (SOLUTION), В КОТОРОМ ДОЛЖНЫ БЫТЬ СОЗДАНЫ ОТДЕЛЬНЫЕ ПРОЕКТЫ (PROJECTS) ДЛЯ КАЖДОЙ ЛАБОРАТОРНОЙ РАБОТЫ. ВО ВСЕХ ПРОЕКТАХ ПОЛЬЗОВАТЕЛЬ ДОЛЖЕН САМ РЕШАТЬ ВЫЙТИ ИЗ ПРОГРАММЫ ИЛИ ПРОДОЛЖИТЬ ВВОД ДАННЫХ. ВСЕ РЕШАЕМЫЕ ЗАДАЧИ ДОЛЖНЫ БЫТЬ РЕАЛИЗОВАНЫ, ИСПОЛЬЗУЯ НЕОБХОДИМЫЕ КЛАССЫ И ОБЪЕКТЫ.**

#### **ВАРИАНТЫ ИНДИВИДУАЛЬНЫХ ЗАДАНИЙ К ЛАБОРАТОРНОЙ РАБОТЕ:**

Использовать контейнер библиотеки *STL*, который будет хранить объекты класса по предметной области, указанной в таблице №4. Класс должен содержать функционал по предметной области. Для контейнера реализовать добавление, удаление элементов, вывод содержимого элементов на экран, редактирование элементов. Необходимо создать удобное пользовательское меню и табличный вывод данных.

Таблица №4 – Типы контейнеров библиотеки *STL* и предметные области индивидуальных заданий

№	Контейнеры	Тема
1.	<i>queue, stack, priority queue</i>	Учет туристических услуг
2.	<i>queue, stack, priority queue</i>	Учет книжной продукции
3.	<i>queue, stack, priority queue</i>	Служба доставки еды
4.	<i>queue, stack, priority queue</i>	Учет пациентов в поликлинике
5.	<i>queue, stack, priority queue</i>	Учет абонентов сотовой связи
6.	<i>queue, stack, priority queue</i>	Учет продуктов питания в столовой
7.	<i>queue, stack, priority queue</i>	Учет беспилотных летательных аппаратов
8.	<i>queue, stack, priority queue</i>	Учет деятельности логистической компании
9.	<i>queue, stack, priority queue</i>	Учет деятельности кинотеатра
10.	<i>queue, stack, priority queue</i>	Учет объектов недвижимости
11.	<i>queue, stack, priority queue</i>	Учет товаров в аптеке
12.	<i>queue, stack, priority queue</i>	Учет услуг в медицинском центре
13.	<i>queue, stack, priority queue</i>	Учет поступлений в библиотеку
14.	<i>queue, stack, priority queue</i>	Служба доставки еды
15.	<i>queue, stack, priority queue</i>	Учет пациентов стоматологической поликлиники
16.	<i>queue, stack, priority queue</i>	Учет клиентов сотовой связи
17.	<i>queue, stack, priority queue</i>	Каталог мобильных телефонов
18.	<i>queue, stack, priority queue</i>	Учет деятельности строительной компании
19.	<i>queue, stack, priority queue</i>	Учет компьютерной техники
20.	<i>queue, stack, priority queue</i>	Учет товаров в магазине
21.	<i>queue, stack, priority queue</i>	Учет объектов недвижимости
22.	<i>queue, stack, priority queue</i>	Учет товаров в продуктовом магазине
23.	<i>queue, stack, priority queue</i>	Учет деятельности медицинской лаборатории
24.	<i>queue, stack, priority queue</i>	Станция технического обслуживания автомобилей
25.	<i>queue, stack, priority queue</i>	Разработка компьютерных игр
26.	<i>queue, stack, priority queue</i>	Разработка программного обеспечения
27.	<i>queue, stack, priority queue</i>	Учет сотрудников логистической компании
28.	<i>queue, stack, priority queue</i>	Обслуживание клиентов банка
29.	<i>queue, stack, priority queue</i>	Учет результатов спортивных соревнований
30.	<i>queue, stack, priority queue</i>	Школа иностранных языков