Министерство высшего образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

ПЕРМСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ (ПНИПУ)

Электротехнический факультет

Дисциплина: основы алгоритмизации и программирования, 2 семестр

**ОТЧЁТ**

Тема: «Шаблоны классов»

Выполнил

Студент РИС-22-2б

Зубов Р.А.

Проверил

Доцент кафедры ИТАС

Полякова О. А.

Пермь 2023

**Постановка задачи**

Задача 1.

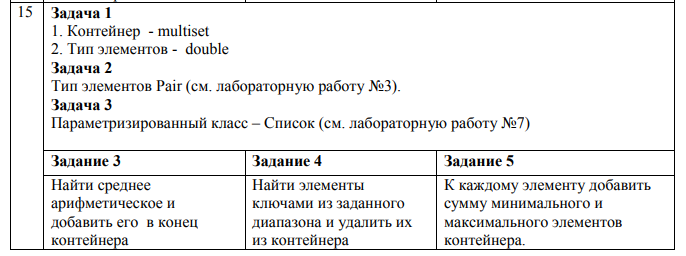
* + 1. Создать ассоциативный контейнер.
    2. Заполнить его элементами стандартного типа (тип указан в варианте).
    3. Добавить элементы в соответствии с заданием
    4. Удалить элементы в соответствии с заданием.
    5. Выполнить задание варианта для полученного контейнера.
    6. Выполнение всех заданий оформить в виде глобальных функций.

Задача 2.

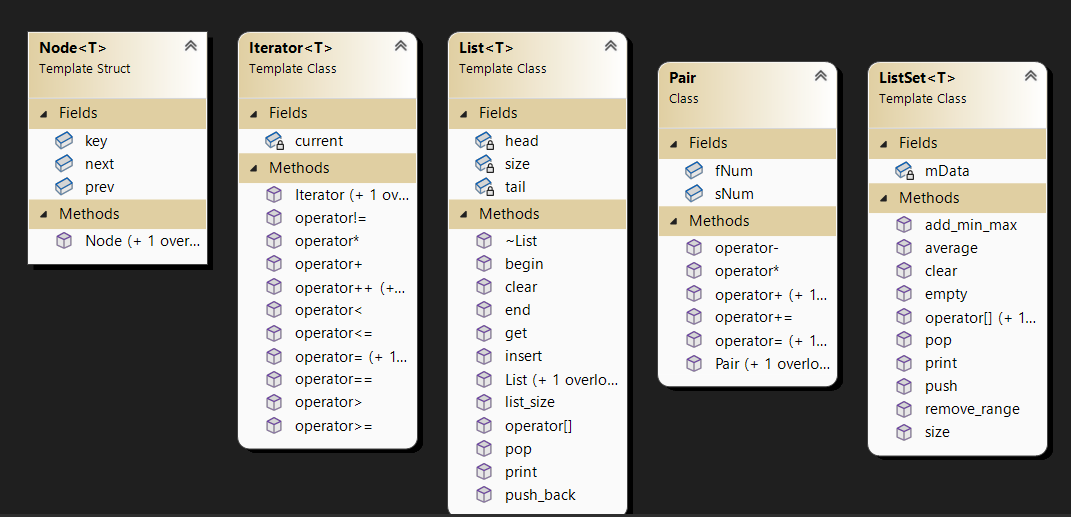
1. Создать ассоциативный контейнер.
2. Заполнить его элементами пользовательского типа (тип указан в варианте). Для пользовательского типа перегрузить необходимые операции.
3. Добавить элементы в соответствии с заданием
4. Удалить элементы в соответствии с заданием.
5. Выполнить задание варианта для полученного контейнера.
6. Выполнение всех заданий оформить в виде глобальных функций.

Задача 3

1. Создать параметризированный класс, используя в качестве контейнера ассоциативный контейнер.
2. Заполнить его элементами.
3. Добавить элементы в соответствии с заданием
4. Удалить элементы в соответствии с заданием.
5. Выполнить задание варианта для полученного контейнера.
6. Выполнение всех заданий оформить в виде методов параметризированного класса.

****

**UML**

****

**Код программы**

**Pair.h:** #pragma once

#include <iostream>

class Pair

{

public:

int fNum = 0;

double sNum = 0;

Pair() {}

Pair(int first, double second) {

fNum = first;

sNum = second;

}

Pair& operator=(const Pair& t);

Pair& operator=(const int& t);

Pair operator\*(Pair& t);

friend std::ostream& operator<<(std::ostream& out, const Pair& t);

friend std::istream& operator>>(std::istream& in, const Pair& t);

};

**Pair.cpp:**

#include "Pair.h"

Pair& Pair::operator=(const Pair& pair) {

this->fNum = pair.fNum;

this->sNum = pair.sNum;

return \*this;

}

Pair& Pair::operator=(const int& t) {

this->fNum = t;

this->sNum = t;

return \*this;

}

Pair Pair::operator\*(Pair& pair) {

Pair new\_pair;

new\_pair.fNum = this->fNum \* pair.fNum;

new\_pair.sNum = this->sNum \* pair.sNum;

return new\_pair;

}

std::ostream& operator<<(std::ostream& out, const Pair& pair) {

out << pair.fNum << " : " << pair.sNum;

return out;

}

std::istream& operator>>(std::istream& in, Pair& pair) {

in >> pair.fNum >> pair.sNum;

return in;

}

**List.h:** #pragma once

#include <cstddef>

#include <iostream>

template <typename T>

struct Node {

T key;

Node<T>\* next = NULL;

Node<T>\* prev = NULL;

Node();

Node(T k, Node<T>\* n, Node<T>\* p);

};

template<typename T>

class Iterator {

template<typename T>

friend class List;

private:

Node<T>\* current = NULL;

public:

Iterator() { current = NULL; }

Iterator(Node<T>\* node) : current(node) {}

bool operator==(const Iterator<T>& other) const { //ñðàâíèâàþò òåêóùèé óçåë

return current == other.current; // è óçåë äðóãîãî èòåðàòîðà

} //

bool operator!=(const Iterator<T>& other) const { // íà ðàâåíñòâî è íåðàâåíñòâî.

return current != other.current;

}

T& operator\*() const { // îïåðàöèÿ ðàçûìåíîâàíèÿ èòåðàòîðà

return current->key;

}

Iterator& operator++() { //++i

current = current->next;

return \*this;

}

Iterator operator++(int) { // i++

Iterator<T> old = \*this;

current = current->next;

return old;

}

Iterator operator+(int n) const { // ïåðåõîä âïðàâî ê n ýëåìåíòó

Iterator<T> it(\*this);

while (n > 0 && it.current != NULL) {

it.current = it.current->next;

n--;

}

return it;

}

};

template<typename T>

class List {

Node<T>\* head;

Node<T>\* tail;

int size;

public:

List() {

head = NULL;

tail = NULL;

size = 0;

}

~List() {

clear();

}

T& get(int index);

T& operator[](int index) {

return get(index); // âîçâðàùàåì ññûëêó íà ýëåìåíò ïî èíäåêñó

}

// Ìåòîä äîáàâëåíèÿ ýëåìåíòà â êîíåö ñïèñêà

void push\_back(T k);

void pop(T key);

int list\_size() {

return size;

}

void clear();

void print() const;

friend std::ostream& operator<<(std::ostream& out, const List<T>& list)

{

list.print();

return out;

}

friend std::istream& operator>>(std::istream& in, List<T>& list)

{

int k;

in >> k;

list.push\_back(k);

return in;

}

Iterator<T> beg() {

return Iterator<T>(head); // âîçâðàùàåì èòåðàòîð íà ïåðâûé ýëåìåíò

}

Iterator<T> end() {

return Iterator<T>(tail->next); // âîçâðàùàåì èòåðàòîð íà ïîñëåäíèé ýëåìåíò

}

friend List<T> operator\*(List<T>& left, List<T>& right) {

List<T> result; // ñîçäàåì íîâûé ñïèñîê

Iterator<T> it1 = left.beg(); // èòåðàòîð ïî ïåðâîìó ñïèñêó

Iterator<T> it2 = right.beg(); // èòåðàòîð ïî âòîðîìó ñïèñêó

while (it1 != left.end() && it2 != right.end()) {

T res = (\*it1) \* (\*it2);

result.push\_back(res); // äîáàâëÿåì ïðîèçâåäåíèå ýëåìåíòîâ â íîâûé ñïèñîê

++it1; // ïåðåõîäèì ê ñëåäóþùåìó ýëåìåíòó ïåðâîãî ñïèñêà

++it2; // ïåðåõîäèì ê ñëåäóþùåìó ýëåìåíòó âòîðîãî ñïèñêà

}

return result; // âîçâðàùàåì íîâûé ñïèñîê

}

};

template<typename T>

Node<T>::Node(T k, Node<T>\* n, Node<T>\* p)

{

key = k;

next = n;

prev = p;

}

template<typename T>

T& List<T>::get(int index)

{

Node<T>\* current = head; // íà÷èíàåì ñ ãîëîâû ñïèñêà

int i = 0;

while (i < index && current->next != NULL) {

current = current->next; // ïåðåõîäèì ê ñëåäóþùåìó óçëó

}

return current->key; // âîçâðàùàåì ññûëêó íà äàííûå óçëà

}

template<typename T>

void List<T>::push\_back(T k)

{

Node<T>\* newNode = new Node<T>(k, NULL, tail); // ñîçäàåì íîâûé óçåë ñ êëþ÷îì key, óêàçàòåëåì íà ñëåäóþùèé óçåë ðàâíûì NULL è óêàçàòåëåì íà ïðåäûäóùèé óçåë ðàâíûì tail

if (tail != NULL) {

tail->next = newNode; // åñëè tail íå ðàâåí NULL, òî óñòàíàâëèâàåì óêàçàòåëü íà ñëåäóþùèé óçåë ó tail íà íîâûé óçåë

}

tail = newNode; // óñòàíàâëèâàåì tail íà íîâûé óçåë

if (head == NULL) {

head = newNode; // åñëè ñïèñîê áûë ïóñòîé, òî óñòàíàâëèâàåì head íà íîâûé óçåë

}

size++;

}

template<typename T>

void List<T>::pop(T key) {

Node<T>\* p = head;

while (p != NULL && p->key != key) { // Ïðîõîäèì ïî ñïèñêó äî íóæíîãî êëþ÷à

p = p->next;

}

if (p != NULL) { // Åñëè íàøëè óçåë ïî ïîçèöèè

if (p->prev != NULL) { // Åñëè óçåë íå ÿâëÿåòñÿ ãîëîâíûì

p->prev->next = p->next;

}

else { // Åñëè óçåë ÿâëÿåòñÿ ãîëîâíûì

head = p->next;

}

if (p->next != NULL) { // Åñëè óçåë íå ÿâëÿåòñÿ õâîñòîâûì

p->next->prev = p->prev;

}

else { // Åñëè óçåë ÿâëÿåòñÿ õâîñòîâûì

tail = p->prev;

}

delete p; // Óäàëÿåì òåêóùèé óçåë èç ïàìÿòè

size--;

}

}

template<typename T>

void List<T>::clear()

{

Node<T>\* p = head;

while (p != NULL) {

Node<T>\* q = p;

p = p->next;

delete q; // Óäàëÿåì òåêóùèé óçåë èç ïàìÿòè

}

head = NULL;

tail = NULL;

size = 0;

}

template<typename T>

void List<T>::print() const

{

Node<T>\* node = head;

while (node != NULL) {

std::cout << node->key << "->";

node = node->next;

}

std::cout << std::endl;

}

**ListSet.h:**

#pragma once

#include <set>

template <typename T>

class ListSet {

public:

void push(const T& value) {

mData.insert(value);

}

void pop() {

auto last\_it = std::prev(mData.end());

mData.erase(last\_it);

}

T& operator[](std::size\_t index) {

auto it = mData.begin();

std::advance(it, index);

return \*it;

}

const T& operator[](std::size\_t index) const {

auto it = mData.begin();

std::advance(it, index);

return \*it;

}

std::size\_t size() const {

return mData.size();

}

bool empty() const {

return mData.empty();

}

void clear() {

mData.clear();

}

void print() const {

for (const auto& item : mData) {

std::cout << item << " | ";

}

std::cout << std::endl;

}

T average() const {

T sum = 0;

for (const auto& item : mData) {

sum += item;

}

return sum / mData.size();

}

void remove\_range(const T& key\_start, const T& key\_end, const T& key) {

auto s\_it = mData.lower\_bound(key\_start);

auto e\_it = mData.upper\_bound(key\_end);

for (auto it = s\_it; it != e\_it; ) {

if (\*it == key) {

it = mData.erase(it++);

}

else {

++it;

}

}

}

void add\_min\_max() {

T min\_val = \*mData.begin();

T max\_val = \*mData.rbegin();

std::multiset<double> buffer;

for (auto& x : mData) {

buffer.insert(x + min\_val + max\_val);

}

mData.clear();

mData.insert(buffer.begin(), buffer.end());

}

private:

std::multiset<T> mData;

};

**Source.cpp:**

#include <iostream>

#include <set>

#include "Pair.h"

#include "ListSet.h"

using namespace std;

template<typename T>

void printSet(multiset<T> S)

{

for (T x : S)

cout << x << " | ";

cout << endl;

}

//Task 1

void task1()

{

multiset<double> set;

for (int i = 0; i < 6; i++)

set.insert((double)rand() / 1000.00);

printSet(set);

//find avg

double sum = 0;

for (auto it = set.begin(); it != set.end(); ++it)

sum += \*it;

set.insert(sum/set.size());

printSet(set);

//find key

int d1, d2;

cout << "Enter range from: "; cin >> d1;

cout << " to: "; cin >> d2;

double key;

cout << "Enter key to delete: "; cin >> key;

auto s\_it = set.begin();

advance(s\_it, d1);

auto e\_it = set.begin();

advance(e\_it, d2);

for (auto it = s\_it; it != e\_it;)

{

if (\*it == key) set.erase(it++);

else ++it;

}

cout << "After deleting: \n";

printSet(set);

//sum max min

cout << "Adding sum to elems: \n";

double MMsum = \*min\_element(set.begin(), set.end())

+ \*max\_element(set.begin(), set.end());

multiset<double> newSet;

for (auto it = set.begin(); it != set.end(); ++it) {

newSet.insert(\*it + MMsum);

}

printSet(newSet);

}

//Task 2

void task2() {

multiset<Pair> set;

set.insert(Pair(1, 1.1));

set.insert(Pair(2, 2.2));

set.insert(Pair(3, 3.3));

set.insert(Pair(4, 4.4));

printSet(set);

//find avg

Pair sum(0, 0.0);

for (auto it = set.begin(); it != set.end(); ++it)

sum = sum + \*it;

sum.fNum /= set.size(); sum.sNum /= set.size();

set.insert(sum);

printSet(set);

//find key

int d1, d2;

cout << "Enter range from: "; cin >> d1;

cout << " to: "; cin >> d2;

Pair key;

cout << "Enter key to delete: "; cin >> key;

auto s\_it = set.begin();

advance(s\_it, d1);

auto e\_it = set.begin();

advance(e\_it, d2);

for (auto it = s\_it; it != e\_it;)

{

if (\*it == key) set.erase(it++);

else ++it;

}

cout << "After deleting: \n";

printSet(set);

//sum max min

cout << "Adding sum to elems: \n";

Pair MMsum = \*min\_element(set.begin(), set.end())

+ \*max\_element(set.begin(), set.end());

multiset<Pair> newSet;

for (auto it = set.begin(); it != set.end(); ++it) {

newSet.insert(\*it + MMsum);

}

printSet(newSet);

}

//Task 3

void task3() {

ListSet<double> List;

List.push(1);

List.push(2);

List.push(3);

List.push(4);

List.print();

List.push(List.average());

List.print();

int d1, d2;

cout << "Range: "; cin >> d1 >> d2;

double key;

cout << "Key: "; cin >> key;

List.remove\_range(d1,d2,key);

List.print();

List.add\_min\_max();

List.print();

}

int main()

{

srand(time(0));

cout << "Results: \n";

cout << "===================\n\n";

cout << "------TASK1------\n";

task1();

/\*cout << "------TASK2------\n";

task2();\*/

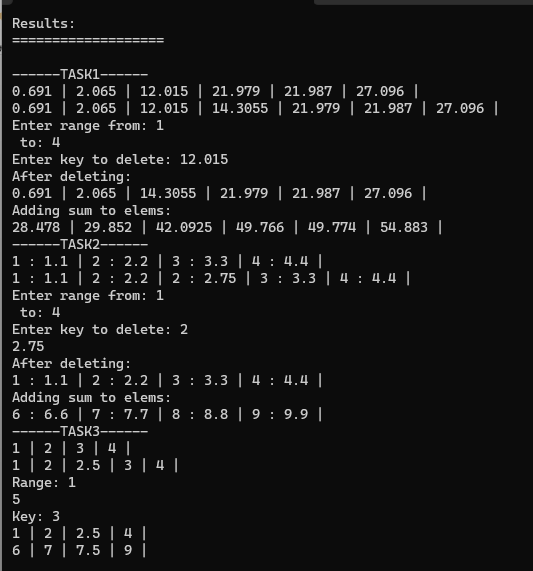
cout << "------TASK3------\n";

task3();

return 0;

}

**Результаты программ**



**Контрольные вопросы**

1. Что представляет собой ассоциативный контейнер?

Ассоциативный контейнер - это контейнер, который содержит упорядоченный набор элементов, представленных в форме пар ключ-значение. Элементы располагаются в контейнере в соответствии с заданным отношением порядка, определенным в компараторе. Ключи уникальны в контейнере, т.е. в контейнере не может быть двух элементов с одинаковым ключом. Основным преимуществом ассоциативных контейнеров является возможность быстрого поиска элемента по ключу благодаря использованию бинарного дерева поиска (например, красно-черного дерева) для хранения элементов. Примерами ассоциативных контейнеров в STL являются map и set.

1. Перечислить ассоциативные контейнеры в библиотеке STL.

set - контейнер, хранящий уникальные элементы в отсортированном порядке. Все элементы в set уникальны, то есть не могут дублироваться.

map - контейнер, хранящий пары ключ-значение, отсортированные по ключу. Ключи в map должны быть уникальными.

multiset - контейнер, хранящий уникальные элементы в отсортированном порядке, но допускающий дублирование элементов.

multimap - контейнер, хранящий пары ключ-значение, отсортированные по

1. Каким образом можно получить доступ к элементам ассоциативного контейнера?

Для доступа к элементам ассоциативного контейнера в STL используются итераторы. В отличие от последовательных контейнеров, в ассоциативных контейнерах элементы хранятся не в порядке их добавления, а в отсортированном порядке на основе ключа. Поэтому для доступа к элементам по индексу, как в последовательных контейнерах, в ассоциативных контейнерах используют итераторы. С помощью итераторов можно получить доступ к ключу и соответствующему ему значению элемента контейнера.

1. Привести примеры методов, используемых в ассоциативных контейнерах.

insert(): добавляет элемент в контейнер.

erase(): удаляет элемент из контейнера.

find(): ищет элемент по заданному ключу.

count(): возвращает количество элементов с заданным ключом в контейнере.

ize(): возвращает количество элементов в контейнере.

mpty(): возвращает значение true, если контейнер пуст, и false в противном случае.

begin(): возвращает итератор, указывающий на первый элемент в контейнере.

end(): возвращает итератор, указывающий на элемент следующий за последним \* элементом контейнера.

ower\_bound(): возвращает итератор на первый элемент в контейнере, не меньший \* заданного ключа.

upper\_bound(): возвращает итератор на первый элемент в контейнере, больший заданного ключа.

equal\_range(): возвращает диапазон элементов в контейнере, соответствующих заданному ключу.

1. Каким образом можно создать контейнер map? Привести примеры.

С помощью конструктора по умолчанию:

map<string, int> myMap;

В данном случае создается пустой контейнер map с ключами типа string и значениями типа int.

С помощью списка инициализации:

map<string, int> myMap = {{"apple", 1}, {"banana", 2}, {"cherry", 3}};

В данном случае создается контейнер map с начальными значениями ключей и значений, которые передаются в список инициализации.

С помощью пары итераторов:

map<std::string, int> myMap(anotherMap.begin(), anotherMap.end());

В данном случае создается контейнер map, который инициализируется парами ключ-значение из другого контейнера, заданного итераторами begin() и end().

С помощью списка пар ключ-значение:

map<std::string, int> myMap = {make\_pair("apple", 1), make\_pair("banana", 2),make\_pair("cherry", 3)};

В данном случае создается контейнер map, который инициализируется парами ключ-значение из списка пар, созданных с помощью функции make\_pair().

1. Каким образом упорядочены элементы в контейнере map по умолчанию? Как изменить порядок на обратный?

Элементы отсортированы в порядке возрастания ключей.

Чтобы изменить порядок, можно определить пользовательскую функцию сравнения, которая будет сравнивать ключи в обратном порядке. Например:

bool compare(int a, int b) {

return a > b;

}

Затем, мы можем создать map следующим образом:

std::map<int, std::string, decltype(compare)\*> myMap(compare);

1. Какие операции определены для контейнера map?

Контейнер map поддерживает операции добавления и удаления элементов, поиска и доступа к элементам по ключу, а также проверки наличия элементов в контейнере. Кроме того, контейнер map поддерживает итераторы для обхода содержимого.

1. Написать функцию для добавления элеентов в контейнер map с помощью функции make\_pair().

template <class T1, class T2>

void addElements(std::map<T1, T2>& m, int n) {

T1 temp1 = 0;

T2 temp2;

for (int i = 0; i < n; i++)

{

cin >> temp2;

m.insert(make\_pair(temp1++, temp2));

}

}

1. Написать функцию для добавления элементов в контейнер map с помощь функции операци прямого доступпа [].

template <class T1, class T2>

void addElements(std::map<T1, T2>& m, T1 key, T2 value) {

m[key] = value;

}

1. Написать функцию для для печати контейнера map с помощью итератора.

template <class T1, class T2>

void printMap(map<T1, T2>& m) {

for (auto it = m.begin(); it != m.end(); ++it) {

cout << it->first << " : " << it->second << endl;

}

}

1. Написать функцию для для печати контейнера map с помощью функции операции прямого доступа [].

template <class T1, class T2>

void printMap(map<T1, T2>& m) {

for (const auto& p : m)

cout << p.first << " : " << m[p.first] << endl;

}

1. Чем отличаются контейнеры map и multimap?

Отличие между map и multimap заключается в том, что map хранит только уникальные ключи и соответствующие значения, тогда как multimap может хранить несколько значений для одного и того же ключа. Другими словами, map — это контейнер с уникальными ключами, а multimap - контейнер с неуникальными ключами.

1. Что представляет собой контейнер set?

Контейнер set - упорядоченное множество уникальных элементов. Он реализован в виде бинарного дерева поиска и обеспечивает быстрый доступ, вставку и удаление элементов в отсортированном порядке.

1. Чем отличаются контейнеры map и set?

Контейнер map предназначен для хранения пары "ключ-значение", где каждый ключ уникален, а контейнер set используется для хранения уникальных элементов, без пары "ключ-значение".

Таким образом, map используется для хранения и доступа к значению по ключу, а set используется для хранения элементов в отсортированном порядке и быстрого поиска элементов по значению.

1. Каким ообразом можно создать контейнер set? Привести примеры.

Создание пустого контейнера с помощью конструктора по умолчанию:

set<int> mySet;

Создание с заданными начальными значениями с помощью списка инициализации:

set<int> mySet = {1, 2, 3, 4};

Создание с помощью диапазона значений другого контейнера:

vector<int> myVec = {1, 2, 3, 4};

set<int> mySet(myVec.begin(), myVec.end());

Создание пустого контейнера с заданным компаратором:

struct Compare {

bool operator()(int a, int b) const {

return a > b;

}

};

set<int, Compare> mySet(Compare());

Создание с заданными начальными значениями и компаратором:

struct Compare {

bool operator()(int a, int b) const {

return a > b;

}

};

set<int, Compare> mySet = {1, 2, 3, 4};

1. Каким образом упорядочены элементы в контейнере set по умолчанию? Как изменить порядок на обратный?

Элементы в контейнере set упорядочены по возрастанию. Чтобы изменить порядок на убывание, можно задать компаратор при создании контейнера, который будет сравнивать элементы в обратном порядке. Например:

#include <functional> // для std::greater

int main() {

set<int,greater<int>> s {5, 2, 7, 1, 8};

// элементы будут упорядочены в порядке убывания

return 0;

}

Здесь std::greater<int> - это функциональный объект, который сравнивает элементы в порядке убывания. Он передается вторым параметром шаблона контейнера set.

1. Какие методы определены для контейнера set?

insert() - добавляет элемент в контейнер

erase() - удаляет элемент из контейнера по значению или по итератору

find() - ищет элемент в контейнере и возвращает итератор на найденный элемент, либо итератор на конец контейнера, если элемент не найден

size() - возвращает количество элементов в контейнере

empty() - возвращает true, если контейнер пуст, иначе – false

clear() - удаляет все элементы из контейнера

1. Написать функцию для добавления элементов в контейнер set.

template <class T1>

void addElements(set<T1>& st, int n) {

for (int i = 0; i < n; i++)

st.insert(T1(rand()));

}

1. Написать функцию для печати контейнера set.

template <class T1>

void printSet(set<T1>& st) {

for (const auto& i : st)

cout << i << endl;

}

1. Чем отличается контейнер set и multiset?

Отличие между ними заключается в том, что set может хранить только уникальные элементы, а multiset может хранить несколько одинаковых элементов.