Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учереждение высшего образования

**«Пермский национальный исследовательский политехнический университет»**

Электротехнический факультет

Кафедра «Информационные технологии и автоматизированные системы»

Направление подготовки: Разработка информационных систем (РИС)

**Лабораторные работы 11-13**

Выполнил студент гр. РИС-24-3б

Караваев Артем Андреевич

Проверил:

Доц. каф. ИТАС

Ольга Андреевна Полякова

г. Пермь, 2024

**Лабораторная работа 11:**

Код программы:

Класс Pair:

#include <iostream>

#include <sstream>

using namespace std;

class Pair{

int x;

double y;

public:

Pair();

Pair(int xx,double yy);

Pair(const Pair&);

~Pair();

int getx();

double gety();

void setx(int xx);

void sety(double yy);

Pair operator+(const Pair&) const;

Pair operator-(const Pair&) const;

Pair operator/(const Pair&) const;

Pair operator\*(const Pair&) const;

bool operator>=(const Pair& p) const;

bool operator<=(const Pair& p) const;

Pair& operator=(const Pair&);

friend ostream& operator<<(ostream&,const Pair&);

friend istream& operator>>(istream&,Pair&);

};

Pair::Pair(){

x=0;

y=0;

}

Pair::Pair(int xx,double yy){

x=xx;

y=yy;

}

Pair::Pair(const Pair& p){

x=p.x;

y=p.y;

}

Pair::~Pair(){

}

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

x=p.x;

y=p.y;

return \*this;

}

int Pair::getx(){

return x;

}

double Pair::gety(){

return y;

}

void Pair::setx(int xx){

x=xx;

}

void Pair::sety(double yy){

y=yy;

}

Pair Pair::operator+(const Pair& p) const

{

return Pair(x+p.x,y+p.y);

}

Pair Pair::operator-(const Pair& p) const

{

return Pair(x-p.x,y-p.y);

}

Pair Pair::operator\*(const Pair& p) const

{

return Pair(x\*p.x,y\*p.y);

}

Pair Pair::operator/(const Pair& p) const

{

return Pair(x/p.x,y/p.y);

}

istream& operator>>(istream& fin,Pair& p){

cout<<"x= ";

fin>>p.x;

cout<<"y= ";

fin>>p.y;

return fin;

}

ostream& operator<<(ostream& out,const Pair& p){

out<<'('<<p.x<<":"<<p.y<<")";

return out;

}

bool Pair::operator>=(const Pair& p) const {

return (x >= p.x) && (y >= p.y);

}

bool Pair::operator<=(const Pair& p) const {

return (x <= p.x) && (y <= p.y);

}

Класс List:

#pragma once

#include <iostream>

#include "iterator.h"

using namespace std;

template <typename T>

class List {

private:

int size;

Node<T>\* head = nullptr;

Node<T>\* tail = nullptr;

public:

List(int s=0,T k=T());

List(const List<T>& a);

~List();

T front();

T back();

void pushback(T data);

void pushfront(T data);

void popback();

void popfront();

bool empty();

List<T>& operator=(const List<T>& a);

T& operator[](int index);

int& operator()();

Iterator<T> first() { return Iterator<T>(head); }

Iterator<T> last() { return Iterator<T>(nullptr); }

Iterator<T> erase(Iterator<T> it){

if (it.elem == nullptr) return it;

Node<T>\* temp=it.elem;

Node<T>\* temp1=temp->next;

if (temp->prev) temp->prev->next=temp->next;

else head = temp->next;

if (temp->next) temp->next->prev=temp->prev;

else tail = temp->prev;

delete temp;

size--;

return temp1;

}

friend ostream& operator<<(ostream& out, const List<T>& a) {

out << endl << "---Вывод элементов списка---" << endl;

Node<T>\* current = a.head;

while (current != nullptr) {

out << current->data << ' ';

current = current->next;

}

out << endl << "---Вывод элементов списка завершён---" << endl;

return out;

}

friend istream& operator>>(istream& in, List<T>& a) {

cout << endl << "---Ввод элементов списка---" << endl;

Node<T>\* current = a.head;

while (current != nullptr) {

in >> current->data;

current = current->next;

}

cout << "---Ввод элементов списка завершён---" << endl;

return in;

}

};

template <typename T>

List<T>::List(int s, T k) {

size = s;

if (size > 0) {

Node<T>\* node = new Node<T>;

node->data = k;

head = node;

tail = node;

for (int i = 1; i < size; ++i) {

Node<T>\* newNode = new Node<T>;

newNode->data = k;

tail->next = newNode;

newNode->prev = tail;

tail = newNode;

}

tail->next = nullptr;

}

}

template <typename T>

List<T>::List(const List<T>& a) {

Node<T>\* node = a.head;

while (node != nullptr) {

pushback(node->data);

node = node->next;

}

}

template <typename T>

List<T>::~List() {

while (head != nullptr) {

popfront();

}

}

template <typename T>

void List<T>::pushfront(T data) {

Node<T>\* newNode = new Node<T>;

newNode->data = data;

if (head == nullptr) {

head = newNode;

tail = newNode;

} else {

newNode->next = head;

head->prev = newNode;

head = newNode;

}

size++;

}

template <typename T>

void List<T>::pushback(T data) {

Node<T>\* newNode = new Node<T>;

newNode->data = data;

if (head == nullptr) {

head = newNode;

tail = newNode;

} else {

tail->next = newNode;

newNode->prev = tail;

tail = newNode;

}

size++;

}

template <typename T>

void List<T>::popback() {

if (tail != nullptr) {

Node<T>\* temp = tail;

tail = tail->prev;

if (tail != nullptr) {

tail->next = nullptr;

} else {

head = nullptr;

}

delete temp;

size--;

}

}

template <typename T>

void List<T>::popfront() {

if (head != nullptr) {

Node<T>\* temp = head;

head = head->next;

if (head != nullptr) {

head->prev = nullptr;

} else {

tail = nullptr;

}

delete temp;

size--;

}

}

template <typename T>

T List<T>::front() {

if (head != nullptr) {

return head->data;

}

}

template <typename T>

T List<T>::back() {

if (tail != nullptr) {

return tail->data;

}

}

template <typename T>

int& List<T>::operator()() {

return size;

}

template <typename T>

List<T>& List<T>::operator=(const List<T>& a) {

if (this != &a) {

while (head != nullptr) {

popfront();

}

Node<T>\* current = a.head;

while (current != nullptr) {

pushback(current->data);

current = current->next;

}

}

return \*this;

}

template <typename T>

T& List<T>::operator[](int index) {

if (index < 0 || index >= size) {

exit(1);

}

Node<T>\* current = head;

for (int i = 0; i < index; ++i) {

current = current->next;

}

return current->data;

}

Класс iterator:

#pragma once

#include <iostream>

using namespace std;

template <typename T>

struct Node {

Node\* prev = nullptr;

Node\* next = nullptr;

T data;

};

template <typename T>

class Iterator {

public:

Node<T>\* elem;

Iterator(Node<T>\* node = nullptr) : elem(node) {}

Iterator() {

elem = nullptr;

}

Iterator(const Iterator& it) {

elem = it.elem;

}

Iterator& operator=(const Iterator& a) {

elem = a.elem;

return \*this;

}

bool operator==(const Iterator& it) const {

return elem == it.elem;

}

bool operator!=(const Iterator& it) const {

return elem != it.elem;

}

Iterator& operator++() {

elem = elem->next;

return \*this;

}

Iterator& operator--() {

elem = elem->prev;

return \*this;

}

Iterator operator+(int a) const {

Iterator temp = \*this;

for (int i = 0; i < a; ++i) {

temp.elem = temp.elem->next;

}

return temp;

}

Iterator operator-(int a) const {

Iterator temp = \*this;

for (int i = 0; i < a; ++i) {

temp.elem = temp.elem->prev;

}

return temp;

}

T& operator\*() const {

return elem->data;

}

};

Функции Main:

#include "list.h"

#include <iostream>

#include <queue>

using namespace std;

template <class T>

List<T> copy\_pqueue\_to\_list(priority\_queue<T> pq) {

List<T> lst;

while (!pq.empty()) {

lst.pushback(pq.top());

pq.pop();

}

return lst;

}

void print\_pqueue(priority\_queue<double>& pq) {

priority\_queue<double> temp = pq;

while (!temp.empty()) {

cout << temp.top() << endl;

temp.pop();

}

cout << endl;

}

template <class T>

priority\_queue<T> copy\_list\_to\_pqueue(List<T> lst) {

priority\_queue<T> pq;

for(auto it=lst.first();it!=lst.last();++it){

pq.push(\*it);

}

return pq;

}

void DobSrAr(priority\_queue<double>& pq) {

if (pq.empty()) return;

List<double> lst = copy\_pqueue\_to\_list(pq);

double sum = 0;

for(auto it=lst.first();it!=lst.last();++it){

sum+=\*it;

}

double average = sum / lst();

lst.pushback(average);

pq = copy\_list\_to\_pqueue(lst);

}

void remove\_key(priority\_queue<double>& pq, double a, double b) {

List<double> lst = copy\_pqueue\_to\_list(pq);

auto it = lst.first();

while (it != lst.last()) {

if (\*it >= a && \*it <= b) {

it = lst.erase(it);

} else {

++it;

}

}

pq = copy\_list\_to\_pqueue(lst);

}

void dob(priority\_queue<double>& pq) {

if (pq.empty()) return;

List<double> lst = copy\_pqueue\_to\_list(pq);

double max\_val = lst.front();

double min\_val = lst.back();

double sum=max\_val+min\_val;

for(auto it=lst.first();it!=lst.last();++it){

\*it+=sum;

}

pq = copy\_list\_to\_pqueue(lst);

}

int main(){

int n;

cout << "Размер очереди = ";

cin >> n;

priority\_queue<double> pq;

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

cout << "Элемент " << i << " : ";

double h;

cin >> h;

pq.push(h);

}

print\_pqueue(pq);

DobSrAr(pq);

print\_pqueue(pq);

double a, b;

cout << "Введите a: "; cin >> a;

cout << "Введите b: "; cin >> b;

remove\_key(pq, a, b);

print\_pqueue(pq);

dob(pq);

print\_pqueue(pq);

return 0;

}

#include <queue>

#include <vector>

#include <iostream>

using namespace std;

void print\_pqueue(priority\_queue<double>& pq) {

priority\_queue<double> temp = pq;

while (!temp.empty()) {

cout << temp.top() << endl;

temp.pop();

}

cout << endl;

}

void DobSrAr(priority\_queue<double>& pq) {

if (pq.empty()) return;

double sum = 0;

int kol = pq.size();

priority\_queue<double> temp = pq;

while (!temp.empty()) {

sum += temp.top();

temp.pop();

}

pq.push(sum / kol);

}

void remove\_key(priority\_queue<double>& pq, double a, double b) {

priority\_queue<double> new\_pq;

while (!pq.empty()) {

double val = pq.top();

pq.pop();

if (val < a || val > b) {

new\_pq.push(val);

}

}

pq = new\_pq;

}

void dob(priority\_queue<double>& pq) {

if (pq.empty()) return;

priority\_queue<double> temp = pq;

double max\_val = temp.top();

double min\_val = max\_val;

while (!temp.empty()) {

min\_val = temp.top();

temp.pop();

}

double sum = max\_val + min\_val;

priority\_queue<double> new\_pq;

while (!pq.empty()) {

new\_pq.push(pq.top() + sum);

pq.pop();

}

pq = new\_pq;

}

int main() {

int n;

cout << "Размер очереди = ";

cin >> n;

priority\_queue<double> pq;

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

cout << "Элемент " << i << " : ";

double h;

cin >> h;

pq.push(h);

}

print\_pqueue(pq);

DobSrAr(pq);

print\_pqueue(pq);

double a, b;

cout << "Введите a: "; cin >> a;

cout << "Введите b: "; cin >> b;

remove\_key(pq, a, b);

print\_pqueue(pq);

dob(pq);

print\_pqueue(pq);

return 0;

}

#include "list.h"

#include <iostream>

using namespace std;

List<double> make\_List(int n){

List<double> lst;

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

cout<<"Элемент списка "<<i<<" : ";

double h;

cin>>h;

lst.pushback(h);

}

return lst;

}

void print\_List(List<double>& lst){

for(auto it=lst.first();it!=lst.last();++it){

cout<<\*it<<endl;

}

cout<<endl;

}

void DobSrAr(List<double>& lst){

double sum=0;

int kol=lst();

for(auto it=lst.first();it!=lst.last();++it){

sum+=\*it;

}

lst.pushback(sum/kol);

}

void remove\_key(List<double>& lst,double a,double b){

auto it = lst.first();

while (it != lst.last()) {

if (\*it >= a && \*it <= b) {

it = lst.erase(it);

} else {

++it;

}

}

}

void dob(List<double>& lst){

double max=lst.front();

double min=lst.front();

for(auto it=lst.first();it!=lst.last();++it){

if(\*it>max){

max=\*it;

}

if(\*it<min){

min=\*it;

}

}

int sum=max+min;

for(auto it=lst.first();it!=lst.last();++it){

\*it+=sum;

}

}

int main(){

int n;

cout<<"Размер списка = ";

cin>>n;

List<double> lst;

lst=make\_List(n);

print\_List(lst);

DobSrAr(lst);

print\_List(lst);

double a,b;

cout<<"Введите a: "; cin>>a;

cout<<"Введите b: "; cin>>b;

remove\_key(lst,a,b);

print\_List(lst);

dob(lst);

print\_List(lst);

return 0;

}

#include <list>

#include <iostream>

#include "pair.h"

using namespace std;

list<Pair> make\_list(int n){

list<Pair> lst;

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

cout<<"Элемент списка "<<i<<" : ";

Pair h;

cin>>h;

lst.push\_back(h);

}

return lst;

}

void print\_list(list<Pair>& lst){

for(const Pair& x:lst){

cout<<x<<endl;

}

cout<<endl;

}

void DobSrAr(list<Pair>& lst){

Pair sum(0,0);

for(const Pair& x:lst){

sum=sum+x;

}

Pair sum2;

sum2.setx(sum.getx()/lst.size());

sum2.sety(sum.gety()/lst.size());

lst.push\_back(sum2);

}

void remove\_key(list<Pair>& lst,Pair a,Pair b){

auto it = lst.begin();

while (it != lst.end()) {

if (\*it >= a && \*it <= b) {

it = lst.erase(it);

} else {

++it;

}

}

}

void dob(list<Pair>& lst){

Pair max=lst.front();

Pair min=lst.front();

for(Pair& x:lst){

if (x.getx() > max.getx() || (x.getx() == max.getx() && x.gety() > max.gety())) {

max = x;

}

if (x.getx() < min.getx() || (x.getx() == min.getx() && x.gety() < min.gety())) {

min = x;

}

}

Pair sum=max+min;

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

\*it=\*it+sum;

}

}

int main(){

int n;

cout<<"Размер списка = ";

cin>>n;

list<Pair> lst;

lst=make\_list(n);

print\_list(lst);

DobSrAr(lst);

print\_list(lst);

Pair a,b;

cout<<"Введите a: "; cin>>a;

cout<<"Введите b: "; cin>>b;

remove\_key(lst,a,b);

print\_list(lst);

dob(lst);

print\_list(lst);

return 0;

}

#include <list>

#include <iostream>

using namespace std;

list<double> make\_list(int n){

list<double> lst;

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

cout<<"Элемент списка "<<i<<" : ";

double h;

cin>>h;

lst.push\_back(h);

}

return lst;

}

void print\_list(list<double>& lst){

for(auto x:lst){

cout<<x<<endl;

}

cout<<endl;

}

void DobSrAr(list<double>& lst){

double sum=0;

int kol=lst.size();

for(auto x:lst){

sum+=x;

}

lst.push\_back(sum/kol);

}

void remove\_key(list<double>& lst,double a,double b){

auto it = lst.begin();

while (it != lst.end()) {

if (\*it >= a && \*it <= b) {

it = lst.erase(it);

} else {

++it;

}

}

}

void dob(list<double>& lst){

int max=lst.front();

int min=lst.front();

for(auto x:lst){

if(x>max){

max=x;

}

if(x<min){

min=x;

}

}

int sum=max+min;

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

\*it+=sum;

}

}

int main(){

int n;

cout<<"Размер списка = ";

cin>>n;

list<double> lst;

lst=make\_list(n);

print\_list(lst);

DobSrAr(lst);

print\_list(lst);

double a,b;

cout<<"Введите a: "; cin>>a;

cout<<"Введите b: "; cin>>b;

remove\_key(lst,a,b);

print\_list(lst);

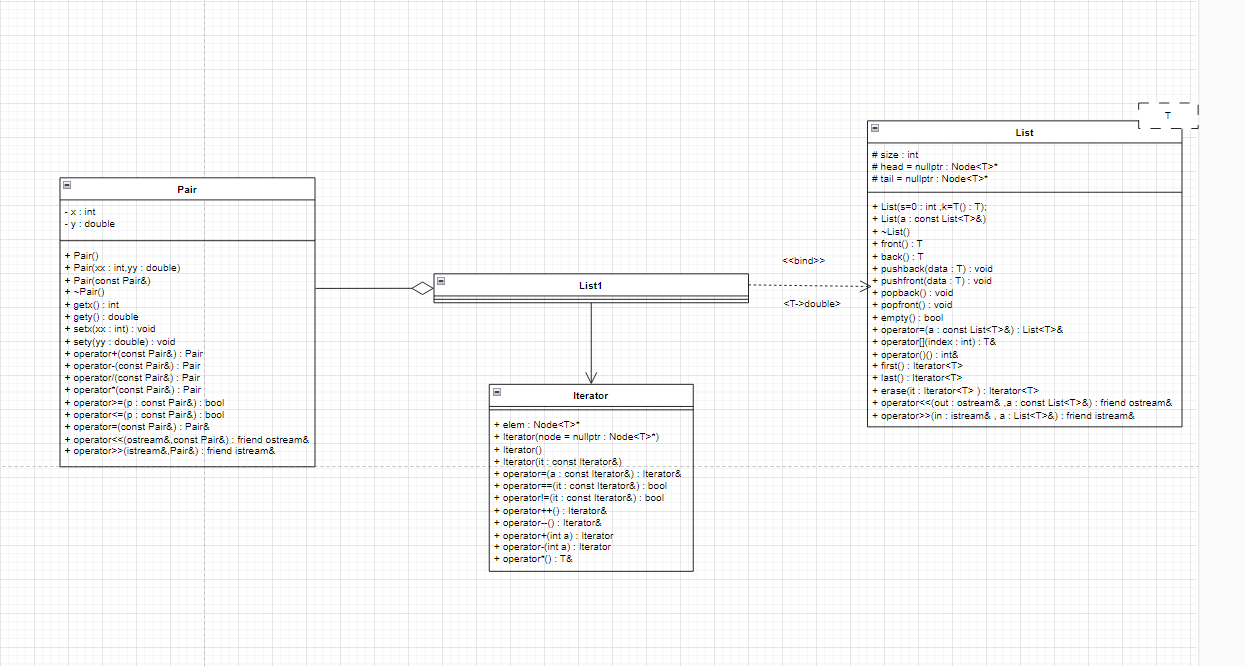
dob(lst);

print\_list(lst);

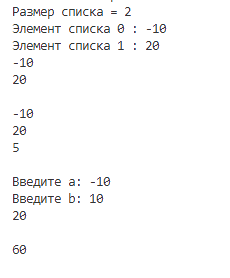
return 0;

}

UML диаграмма:



Входные и выходные данные:



Ответы на вопросы:

1. Контейнеры: Это структуры данных, которые хранят коллекции объектов. Примеры включают vector, list, deque, set, map и т.д. Итераторы: Это объекты, которые позволяют перемещаться по элементам внутри контейнера. Они предоставляют способ доступа и манипулирования элементами в универсальной форме. Алгоритмы: Это функции, которые выполняют различные операции над контейнерами, такие как поиск, сортировка, копирование, преобразование и т.д. Они предназначены для работы с различными типами контейнеров с использованием итераторов. Функциональные объекты (Функторы): Это объекты, которые можно вызывать, как если бы они были функциями. Они часто используются в качестве аргументов для алгоритмов, чтобы настроить их поведение. Аллокаторы: Они отвечают за управление выделением памяти для контейнеров.
2. Последовательные контейнеры: vector, deque, list, forward\_list, array. Ассоциативные контейнеры: set, multiset, map, multimap. Неупорядоченные ассоциативные контейнеры: unordered\_set, unordered\_multiset, unordered\_map, unordered\_multimap. Адаптеры контейнеров: stack, queue, priority\_queue.
3. Включить соответствующий заголовочный файл: Каждый контейнер, алгоритм и т. д. STL имеет свой заголовочный файл. Например, для использования vector нужно включить #include <vector>. Для list нужно #include <list>, для map нужно #include <map> и т.д. Указать пространство имен std: Контейнеры STL, как и большинство элементов стандартной библиотеки C++, находятся в пространстве имен std. Можно использовать using namespace std; в начале файла (хотя это может быть не рекомендовано в крупных проектах) или явно указывать пространство имен при каждом использовании контейнера, например std::vector<int> myVector;
4. Итератор - это обобщенный указатель, используемый для доступа к элементам в контейнере (например, vector, list, map и т.д.) или в последовательности. Можно представить его как указатель на элемент контейнера, но с более мощными и универсальными свойствами.

#include <iostream>

#include <vector>

int main() {

std::vector<int> numbers = {1, 2, 3, 4, 5};

// Получаем итератор на начало вектора

std::vector<int>::iterator it = numbers.begin();

// Перебираем вектор с помощью итератора

while (it != numbers.end()) {

std::cout << \*it << " "; // Выводим значение элемента, на который указывает итератор

++it; // Переходим к следующему элементу

}

std::cout << std::endl;

return 0;

}

1. Разыменование (\*it): Получение значения элемента, на который указывает итератор. Инкремент (++it, it++): Переход к следующему элементу контейнера. Декремент (--it, it--): Переход к предыдущему элементу контейнера (только для bidirectional и random access итераторов). Сравнение (it1 == it2, it1 != it2): Проверка, указывают ли два итератора на один и тот же элемент (или на конец контейнера). Присваивание (it1 = it2): Копирование значения итератора (заставляя it1 указывать на то же место, что и it2). Арифметические операции (+, -, +=, -=): Доступно только для random access итераторов. Позволяет перемещаться на несколько элементов вперед или назад. Пример: it + 5 (переместиться на 5 элементов вперед). Оператор доступа к элементу (it[n]): Доступно только для random access итераторов. Эквивалентно \*(it + n).

While:

#include <iostream>

#include <vector>

int main() {

std::vector<int> numbers = {1, 2, 3, 4, 5};

std::vector<int>::iterator it = numbers.begin(); // Получаем итератор на начало

while (it != numbers.end()) { // Пока итератор не достиг конца

std::cout << \*it << " ";

++it; // Переходим к следующему элементу

}

std::cout << std::endl;

return 0;

}

For:

#include <iostream>

#include <vector>

int main() {

std::vector<int> numbers = {1, 2, 3, 4, 5};

for (std::vector<int>::iterator it = numbers.begin(); it != numbers.end(); ++it) {

std::cout << \*it << " ";

}

std::cout << std::endl;

return 0;

}

1. Input Iterator: Итератор, который поддерживает только чтение значений и перемещение вперед (++). Используется для чтения последовательности данных. Output Iterator: Итератор, который поддерживает только запись значений и перемещение вперед (++). Используется для записи последовательности данных. Forward Iterator: Итератор, который поддерживает чтение, запись и перемещение вперед (++). Можно проходить по последовательности несколько раз, сохраняя позицию. Bidirectional Iterator: Итератор, который поддерживает чтение, запись и перемещение вперед и назад (++, --). Можно проходить по последовательности в обоих направлениях. Random Access Iterator: Итератор, который поддерживает чтение, запись, перемещение вперед и назад, а также произвольный доступ к элементам с помощью арифметических операций (+, -, +=, -=, []).
2. begin(): Возвращает итератор, указывающий на первый элемент контейнера. end(): Возвращает итератор, указывающий на позицию "за концом" последнего элемента контейнера. size(): Возвращает количество элементов в контейнере. empty(): Возвращает true, если контейнер пуст (не содержит элементов), и false в противном случае. clear(): Удаляет все элементы из контейнера. swap(container): Меняет содержимое текущего контейнера с содержимым другого контейнера того же типа. Оператор присваивания (=): Копирует содержимое одного контейнера в другой. max\_size(): Возвращает теоретически максимальное количество элементов, которое может вместить контейнер. Это число может быть ограничено доступной памятью. Тип элементов: Все контейнеры явно определяют тип элементов, которые они хранят (например, std::vector<int>).
3. Произвольный доступ к элементам ([] или at()): Эффективность O(1). vector хранит элементы в непрерывном блоке памяти, подобно массиву. Поэтому можно получить доступ к любому элементу по его индексу за константное время. Добавление элемента в конец (push\_back()): Обычно эффективность O(1) (амортизированное). Добавление в конец обычно выполняется быстро, если есть свободное место в выделенном блоке памяти. Если блок памяти заполнен, vector должен выделить новый блок большего размера, скопировать в него все элементы, и освободить старый блок. Эта операция может быть затратной, но она происходит относительно редко, поэтому эффективность считается амортизированной O(1). Получение размера (size()): Эффективность O(1). vector отслеживает количество элементов, поэтому получение размера занимает константное время. Почему: vector реализован как динамический массив, обеспечивающий непрерывное хранение элементов в памяти. Это делает доступ к элементам по индексу очень быстрым.
4. Вставка и удаление элементов в любом месте (insert(), erase()): Эффективность O(1) при наличии итератора на место вставки/удаления. list реализован как двусвязный список. Вставка или удаление элемента требует только изменения указателей на соседние элементы, без перемещения других элементов. Добавление и удаление элементов в начале или в конце (push\_front(), push\_back(), pop\_front(), pop\_back()): Эффективность O(1). Добавление или удаление в начале или в конце списка требует только изменения указателей на первый и последний элементы. Почему: list реализован как двусвязный список, где каждый элемент хранит указатели на предыдущий и следующий элементы. Это позволяет быстро вставлять и удалять элементы, не затрагивая другие элементы в списке.
5. Добавление и удаление элементов в начале и в конце (push\_front(), push\_back(), pop\_front(), pop\_back()): Эффективность O(1). deque разработан для эффективной работы с обоими концами. Произвольный доступ к элементам ([] или at()): Эффективность O(1). Хотя deque не хранит элементы в одном непрерывном блоке памяти, как vector, он организован в виде массива блоков, что позволяет выполнять произвольный доступ за константное время. Почему: deque (double-ended queue) реализован как массив блоков памяти. Это позволяет быстро добавлять и удалять элементы с обоих концов, а также обеспечивает относительно быстрый произвольный доступ. deque обеспечивает компромисс между эффективностью vector (произвольный доступ) и list (вставка/удаление в середине).
6. Доступ к элементам: at(index): Возвращает элемент по указанному индексу с проверкой выхода за границы. operator[](index): Возвращает элемент по указанному индексу (без проверки границ). front(): Возвращает ссылку на первый элемент. back(): Возвращает ссылку на последний элемент. data(): Возвращает указатель на внутренний массив. Итераторы: begin(): Возвращает итератор на первый элемент. end(): Возвращает итератор на позицию "за концом" последнего элемента. rbegin(): Возвращает обратный итератор на последний элемент. rend(): Возвращает обратный итератор на позицию "перед началом" первого элемента. cbegin(), cend(), crbegin(), crend(): Константные версии соответствующих итераторов. Емкость: empty(): Проверяет, пуст ли контейнер. size(): Возвращает количество элементов. max\_size(): Возвращает максимальное количество элементов, которое может хранить контейнер. capacity(): Возвращает текущий размер выделенной памяти. reserve(n): Запрашивает выделение памяти для хранения минимум n элементов. Изменение контейнера: clear(): Удаляет все элементы. insert(pos, value): Вставляет один элемент value перед итератором pos. insert(pos, n, value): Вставляет n копий элемента value перед итератором pos. insert(pos, first, last): Вставляет элементы из диапазона [first, last) перед итератором pos. emplace(pos, args...): Вставляет новый элемент, сконструированный на месте, перед итератором pos. erase(pos): Удаляет элемент, на который указывает итератор pos. erase(first, last): Удаляет элементы из диапазона [first, last). push\_back(value): Добавляет элемент в конец. emplace\_back(args...): Добавляет элемент в конец, сконструированный на месте. pop\_back(): Удаляет последний элемент. resize(n): Изменяет размер контейнера до n элементов. resize(n, value): Изменяет размер контейнера до n элементов, инициализируя новые элементы значением value. swap(other): Меняет содержимое с другим вектором. Операторы: operator=: Оператор присваивания. operator==, operator!=, operator<, operator<=, operator>, operator>=: Операторы сравнения.
7. Доступ к элементам: front(): Возвращает ссылку на первый элемент. back(): Возвращает ссылку на последний элемент. Итераторы: begin(): Возвращает итератор на первый элемент. end(): Возвращает итератор на позицию "за концом" последнего элемента. rbegin(): Возвращает обратный итератор на последний элемент. rend(): Возвращает обратный итератор на позицию "перед началом" первого элемента. cbegin(), cend(), crbegin(), crend(): Константные версии соответствующих итераторов. Емкость: empty(): Проверяет, пуст ли контейнер. size(): Возвращает количество элементов. max\_size(): Возвращает максимальное количество элементов, которое может хранить контейнер. Изменение контейнера: clear(): Удаляет все элементы. insert(pos, value): Вставляет один элемент value перед итератором pos. insert(pos, n, value): Вставляет n копий элемента value перед итератором pos. insert(pos, first, last): Вставляет элементы из диапазона [first, last) перед итератором pos. emplace(pos, args. erase(pos): Удаляет элемент, на который указывает итератор pos. erase(first, last): Удаляет элементы из диапазона [first, last). push\_back(value): Добавляет элемент в конец. emplace\_back(args...): Добавляет элемент в конец, сконструированный на месте. pop\_back(): Удаляет последний элемент. push\_front(value): Добавляет элемент в начало. emplace\_front(args...): Добавляет элемент в начало, сконструированный на месте. pop\_front(): Удаляет первый элемент. resize(n): Изменяет размер контейнера до n элементов. resize(n, value): Изменяет размер контейнера до n элементов, инициализируя новые элементы значением value. swap(other): Меняет содержимое с другим списком. splice(pos, other): Перемещает все элементы из other в текущий список перед итератором pos. splice(pos, other, it): Перемещает один элемент из other, на который указывает итератор it, в текущий список перед итератором pos. splice(pos, other, first, last): Перемещает элементы из диапазона [first, last) в other в текущий список перед итератором pos. remove(value): Удаляет все элементы, равные value. remove\_if(predicate): Удаляет все элементы, для которых predicate возвращает true. unique(): Удаляет смежные дубликаты. unique(binary\_predicate): Удаляет смежные элементы, для которых binary\_predicate возвращает true. merge(other): Объединяет отсортированный список other с текущим отсортированным списком. merge(other, comp): Объединяет отсортированный список other с текущим отсортированным списком, используя компаратор comp. sort(): Сортирует элементы списка. sort(comp): Сортирует элементы списка, используя компаратор comp. reverse(): Переворачивает порядок элементов в списке. Операторы: operator=: Оператор присваивания. operator==, operator!=, operator<, operator<=, operator>, operator>=: Операторы сравнения.
8. Доступ к элементам: at(index): Возвращает элемент по указанному индексу с проверкой выхода за границы. operator[](index): Возвращает элемент по указанному индексу (без проверки границ). front(): Возвращает ссылку на первый элемент. back(): Возвращает ссылку на последний элемент. Итераторы: begin(): Возвращает итератор на первый элемент. end(): Возвращает итератор на позицию "за концом" последнего элемента. rbegin(): Возвращает обратный итератор на последний элемент. rend(): Возвращает обратный итератор на позицию "перед началом" первого элемента. cbegin(), cend(), crbegin(), crend(): Константные версии соответствующих итераторов. Емкость: empty(): Проверяет, пуст ли контейнер. size(): Возвращает количество элементов. max\_size(): Возвращает максимальное количество элементов, которое может хранить контейнер. Изменение контейнера: clear(): Удаляет все элементы. insert(pos, value): Вставляет один элемент value перед итератором pos. insert(pos, n, value): Вставляет n копий элемента value перед итератором pos. insert(pos, first, last): Вставляет элементы из диапазона [first, last) перед итератором pos. emplace(pos, args...): Вставляет новый элемент, сконструированный на месте, перед итератором pos. erase(pos): Удаляет элемент, на который указывает итератор pos. erase(first, last): Удаляет элементы из диапазона [first, last). push\_back(value): Добавляет элемент в конец. emplace\_back(args...): Добавляет элемент в конец, сконструированный на месте. pop\_back(): Удаляет последний элемент. push\_front(value): Добавляет элемент в начало. emplace\_front(args...): Добавляет элемент в начало, сконструированный на месте. pop\_front(): Удаляет первый элемент. resize(n): Изменяет размер контейнера до n элементов. resize(n, value): Изменяет размер контейнера до n элементов, инициализируя новые элементы значением value. swap(other): Меняет содержимое с другим двусторонней очередью. Операторы: operator=: Оператор присваивания. operator==, operator!=, operator<, operator<=, operator>, operator>=: Операторы сравнения.

15,16. #include <iostream>

#include <vector>

int main() {

// Пример vector

std::vector<int> myVector = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

// Удалить элементы с индекса 2 по 5 (включительно)

// erase принимает диапазон, поэтому передаем итераторы на начало и конец диапазона

myVector.erase(myVector.begin() + 2, myVector.begin() + 6);

// Удалить последний элемент

if (!myVector.empty()) {

myVector.pop\_back();

}

// Вывод для проверки

for (int num : myVector) {

std::cout << num << " ";

}

std::cout << std::endl;

return 0;

}

17,18. #include <iostream>

#include <list>

int main() {

// Пример list

std::list<int> myList = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

// Удалить элементы со 2 по 5 (включительно)

auto it = myList.begin();

std::advance(it, 2); // Перемещаем итератор на 2-й элемент (индекс 2)

auto itEnd = it;

std::advance(itEnd, 4); // Перемещаем итератор на 4 элемента вперед (до 6го элемента, индекс 5)

++itEnd; // Перемещаем itEnd на следующий элемент после 5го (индекс 6), т.к. erase удаляет до, но не включая последний элемент

myList.erase(it, itEnd);

// Удалить последний элемент

if (!myList.empty()) {

myList.pop\_back();

}

// Вывод для проверки

for (int num : myList) {

std::cout << num << " ";

}

std::cout << std::endl;

return 0;

}

19,20. #include <iostream>

#include <deque>

int main() {

// Пример deque

std::deque<int> myDeque = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

// Удалить элементы с индекса 2 по 5 (включительно)

myDeque.erase(myDeque.begin() + 2, myDeque.begin() + 6);

// Удалить последний элемент

if (!myDeque.empty()) {

myDeque.pop\_back();

}

// Вывод для проверки

for (int num : myDeque) {

std::cout << num << " ";

}

std::cout << std::endl;

return 0;

}

21. #include <iostream>

#include <vector>

#include <list>

#include <deque>

// Шаблон функции для печати последовательного контейнера любого типа

template <typename Container>

void printContainer(const Container& container) {

// typename Container::value\_type - тип элемента в контейнере

// auto - автоматически выводит тип переменной

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

std::cout << \*it << " ";

}

std::cout << std::endl;

}

int main() {

std::vector<int> myVector = {1, 2, 3, 4, 5};

std::list<std::string> myList = {"apple", "banana", "cherry"};

std::deque<double> myDeque = {1.1, 2.2, 3.3};

std::cout << "Vector: ";

printContainer(myVector);

std::cout << "List: ";

printContainer(myList);

std::cout << "Deque: ";

printContainer(myDeque);

return 0;

}

22. Адаптеры контейнеров - это типы, которые предоставляют другой интерфейс для существующих контейнеров. Они не являются полноценными контейнерами в том смысле, что не хранят данные напрямую, а скорее используют существующие контейнеры для реализации своей функциональности. Адаптеры контейнеров ограничивают интерфейс базового контейнера, предоставляя только определенный набор операций, которые соответствуют определенной структуре данных.

23. stack<int> s: Это объявление создает стек s, который хранит элементы типа int и использует контейнер deque в качестве базового контейнера по умолчанию. То есть, фактически, stack использует std::deque<int> для хранения элементов. deque является хорошим выбором по умолчанию, поскольку он обеспечивает эффективное добавление и удаление элементов с обоих концов. stack<int, list<int>> s: Это объявление создает стек s, который хранит элементы типа int и явно указывает, что в качестве базового контейнера следует использовать list. То есть, stack использует std::list<int> для хранения элементов. Первое объявление позволяет компилятору выбрать базовый контейнер, который обычно является deque, в то время как второе объявление явно указывает list в качестве базового контейнера.

24. empty(): Возвращает true, если стек пуст (не содержит элементов), и false в противном случае. size(): Возвращает количество элементов в стеке. top(): Возвращает ссылку на верхний элемент стека (последний добавленный элемент). Не удаляет элемент из стека. push(value): Добавляет элемент value на вершину стека. pop(): Удаляет верхний элемент стека.

25. empty(): Возвращает true, если очередь пуста (не содержит элементов), и false в противном случае. size(): Возвращает количество элементов в очереди. front(): Возвращает ссылку на первый элемент в очереди (элемент, который будет извлечен следующим). Не удаляет элемент из очереди. back(): Возвращает ссылку на последний элемент в очереди (элемент, добавленный последним). Не удаляет элемент из очереди. push(value): Добавляет элемент value в конец очереди. pop(): Удаляет первый элемент из очереди.

26. queue (обычная очередь): Элементы извлекаются в порядке FIFO (First-In, First-Out - "первым пришел, первым ушел"). Это означает, что элементы добавляются в конец очереди (push()) и удаляются из начала очереди (pop()). Метод front() возвращает элемент в начале очереди. Порядок элементов определяется моментом их добавления в очередь. priority\_queue (очередь с приоритетами): Элементы извлекаются в соответствии с их приоритетом. По умолчанию, элемент с наибольшим значением имеет наивысший приоритет и будет извлечен первым. При добавлении элемента в очередь (push()) его позиция в очереди определяется его приоритетом. Метод top() возвращает элемент с наивысшим приоритетом (который будет извлечен следующим). Для определения приоритета элементов можно использовать компаратор. Порядок извлечения элементов: В queue элементы извлекаются в порядке их добавления (FIFO - "первым вошел, первым вышел"). В priority\_queue элементы извлекаются в порядке их приоритета (обычно сначала элементы с наибольшим значением). Как добавляются элементы: В queue элементы добавляются в конец. В priority\_queue элементы добавляются таким образом, чтобы поддерживать порядок приоритетов. Как получить следующий элемент: В queue используется front() для получения следующего элемента. В priority\_queue используется top() для получения элемента с наивысшим приоритетом. Основная идея: queue представляет собой простую очередь, в которой важен порядок поступления элементов. priority\_queue представляет собой очередь, в которой важно, какой элемент нужно обработать следующим, исходя из его приоритета.

27. #include <iostream>

#include <stack>

void removeElement(std::stack<int>& s, int index) {

if (index < 0 || index >= s.size()) {

std::cout << "Invalid index" << std::endl;

return;

}

std::stack<int> tempStack;

int currentSize = s.size();

for (int i = 0; i < currentSize - 1 - index; ++i) {

tempStack.push(s.top());

s.pop();

}

s.pop(); // Удаляем элемент с заданным номером

while (!tempStack.empty()) {

s.push(tempStack.top());

tempStack.pop();

}

}

int main() {

std::stack<int> myStack;

myStack.push(10);

myStack.push(20);

myStack.push(30);

myStack.push(40);

std::cout << "Stack before removal:" << std::endl;

std::stack<int> temp = myStack; // Создаем временную копию стека для вывода

while(!temp.empty()) {

std::cout << temp.top() << std::endl;

temp.pop();

}

removeElement(myStack, 1); // Удаляем элемент с номером 1 (считая с вершины)

std::cout << "\nStack after removal:" << std::endl;

temp = myStack; // Обновляем временную копию

while(!temp.empty()) {

std::cout << temp.top() << std::endl;

temp.pop();

}

return 0;

}

28. #include <iostream>

#include <queue>

void removeElement(std::queue<int>& q, int index) {

if (index < 0 || index >= q.size()) {

std::cout << "Invalid index" << std::endl;

return;

}

std::queue<int> tempQueue;

int currentSize = q.size();

for (int i = 0; i < index; ++i) {

tempQueue.push(q.front());

q.pop();

}

q.pop(); // Удаляем элемент с заданным номером

while (!tempQueue.empty()) {

q.push(tempQueue.front());

tempQueue.pop();

}

// Возвращаем элементы обратно в исходный порядок в основной queue

for (int i = 0; i < currentSize - index - 1; ++i) {

q.push(q.front());

q.pop();

}

}

int main() {

std::queue<int> myQueue;

myQueue.push(10);

myQueue.push(20);

myQueue.push(30);

myQueue.push(40);

std::cout << "Queue before removal:" << std::endl;

std::queue<int> temp = myQueue; // Создаем временную копию очереди для вывода

while(!temp.empty()) {

std::cout << temp.front() << " ";

temp.pop();

}

std::cout << std::endl;

removeElement(myQueue, 1); // Удаляем элемент с номером 1

std::cout << "\nQueue after removal:" << std::endl;

temp = myQueue; // Обновляем временную копию

while(!temp.empty()) {

std::cout << temp.front() << " ";

temp.pop();

}

std::cout << std::endl;

return 0;

}

29. #include <iostream>

#include <stack>

template <typename T>

void printStack(std::stack<T> s) { // Принимаем копию стека

std::cout << "Stack (top to bottom): ";

while (!s.empty()) {

std::cout << s.top() << " ";

s.pop();

}

std::cout << std::endl;

}

int main() {

std::stack<int> myStack;

myStack.push(10);

myStack.push(20);

myStack.push(30);

printStack(myStack);

return 0;

}

30. #include <iostream>

#include <queue>

template <typename T>

void printQueue(std::queue<T> q) { // Принимаем копию очереди

std::cout << "Queue (front to back): ";

while (!q.empty()) {

std::cout << q.front() << " ";

q.pop();

}

std::cout << std::endl;

}

int main() {

std::queue<int> myQueue;

myQueue.push(10);

myQueue.push(20);

myQueue.push(30);

printQueue(myQueue);

return 0;

}

**Лабораторная работа 12:**

Код программы:

Класс Pair:

#include <iostream>

#include <sstream>

using namespace std;

class Pair{

int x;

double y;

public:

Pair();

Pair(int xx,double yy);

Pair(const Pair&);

~Pair();

int getx();

double gety();

void setx(int xx);

void sety(double yy);

Pair operator+(const Pair&) const;

Pair operator-(const Pair&) const;

Pair operator/(const Pair&) const;

Pair operator\*(const Pair&) const;

bool operator<(const Pair&) const;

bool operator>=(const Pair& p) const;

bool operator<=(const Pair& p) const;

Pair& operator=(const Pair&);

friend ostream& operator<<(ostream&,const Pair&);

friend istream& operator>>(istream&,Pair&);

};

Pair::Pair(){

x=0;

y=0;

}

Pair::Pair(int xx,double yy){

x=xx;

y=yy;

}

Pair::Pair(const Pair& p){

x=p.x;

y=p.y;

}

Pair::~Pair(){

}

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

x=p.x;

y=p.y;

return \*this;

}

int Pair::getx(){

return x;

}

double Pair::gety(){

return y;

}

void Pair::setx(int xx){

x=xx;

}

void Pair::sety(double yy){

y=yy;

}

Pair Pair::operator+(const Pair& p) const

{

return Pair(x+p.x,y+p.y);

}

Pair Pair::operator-(const Pair& p) const

{

return Pair(x-p.x,y-p.y);

}

Pair Pair::operator\*(const Pair& p) const

{

return Pair(x\*p.x,y\*p.y);

}

Pair Pair::operator/(const Pair& p) const

{

return Pair(x/p.x,y/p.y);

}

istream& operator>>(istream& fin,Pair& p){

cout<<"x= ";

fin>>p.x;

cout<<"y= ";

fin>>p.y;

return fin;

}

ostream& operator<<(ostream& out,const Pair& p){

out<<'('<<p.x<<":"<<p.y<<")";

return out;

}

bool Pair::operator>=(const Pair& p) const {

return (x >= p.x) && (y >= p.y);

}

bool Pair::operator<=(const Pair& p) const {

return (x <= p.x) && (y <= p.y);

}

bool Pair::operator<(const Pair& p) const{

return (x < p.x) || (x==p.x)&& (y < p.y);

}

Класс List:

#pragma once

#include <iostream>

#include <set>

using namespace std;

template <typename T>

class List {

private:

int size;

multiset<T> ms;

public:

List():size(0){};

List(int n);

~List();

void Print();

void make\_multiset(int n);

void DobSrAr();

void remove\_key(T,T);

void dob();

};

template <typename T>

List<T>::List(int n) {

size=n;

}

template <typename T>

List<T>::~List() {

}

template <typename T>

void List<T>:: Print(){

for (auto x : ms) {

cout << x << endl;

}

cout << endl;

}

template <typename T>

void List<T>::make\_multiset(int n) {

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

cout << "Элемент multiset " << i << " : ";

T h;

cin >> h;

ms.insert(h);

}

}

template <typename T>

void List<T>::DobSrAr() {

if (ms.empty()) return;

double sum=0;

for (auto x : ms) {

sum+=x;

}

double average = sum / ms.size();

ms.insert(average);

}

template <typename T>

void List<T>::remove\_key(T a, T b) {

auto it = ms.begin();

while (it != ms.end()) {

if (\*it >= a && \*it <= b) {

it = ms.erase(it);

} else {

++it;

}

}

}

template <typename T>

void List<T>::dob() {

if (ms.empty()) return;

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

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

T sum = min\_val + max\_val;

multiset<T> new\_ms;

for (auto x : ms) {

new\_ms.insert(x + sum);

}

ms = new\_ms;

}

Функции Main:

#include <iostream>

#include "list.h"

using namespace std;

int main() {

int n;

cout << "Размер multiset = ";

cin >> n;

List<double> lst(n);

lst.make\_multiset(n);

lst.Print();

lst.DobSrAr();

lst.Print();

double a, b;

cout << "Введите a: "; cin >> a;

cout << "Введите b: "; cin >> b;

lst.remove\_key(a, b);

lst.Print();

lst.dob();

lst.Print();

return 0;

}

#include <iostream>

#include <set>

#include "pair.h"

using namespace std;

multiset<Pair> make\_multiset(int n) {

multiset<Pair> ms;

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

cout << "Элемент multiset " << i << " : ";

Pair h;

cin >> h;

ms.insert(h);

}

return ms;

}

void print\_multiset(multiset<Pair>& ms) {

for (auto x : ms) {

cout << x << endl;

}

cout << endl;

}

void DobSrAr(multiset<Pair>& ms) {

if (ms.empty()) return;

Pair sum;

for (auto x : ms) {

sum=sum+x;

}

sum.setx(sum.getx()/ms.size());

sum.sety(sum.gety()/ms.size());

ms.insert(sum);

}

void remove\_key(multiset<Pair>& ms, Pair a, Pair b) {

auto it = ms.begin();

while (it != ms.end()) {

if (\*it>=a && \*it <= b) {

it = ms.erase(it);

} else {

++it;

}

}

}

void dob(multiset<Pair>& ms) {

if (ms.empty()) return;

Pair min\_val = \*ms.begin();

Pair max\_val = \*ms.rbegin();

Pair sum = min\_val + max\_val;

multiset<Pair> new\_ms;

for (auto x : ms) {

new\_ms.insert(x + sum);

}

ms = new\_ms;

}

int main() {

int n;

cout << "Размер multiset = ";

cin >> n;

multiset<Pair> ms = make\_multiset(n);

print\_multiset(ms);

DobSrAr(ms);

print\_multiset(ms);

Pair a, b;

cout << "Введите a: "; cin >> a;

cout << "Введите b: "; cin >> b;

remove\_key(ms, a, b);

print\_multiset(ms);

dob(ms);

print\_multiset(ms);

return 0;

}

#include <iostream>

#include <set>

#include <numeric>

using namespace std;

multiset<double> make\_multiset(int n) {

multiset<double> ms;

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

cout << "Элемент multiset " << i << " : ";

double h;

cin >> h;

ms.insert(h);

}

return ms;

}

void print\_multiset(multiset<double>& ms) {

for (auto x : ms) {

cout << x << endl;

}

cout << endl;

}

void DobSrAr(multiset<double>& ms) {

if (ms.empty()) return;

double sum=0;

for (auto x : ms) {

sum+=x;

}

double average = sum / ms.size();

ms.insert(average);

}

void remove\_key(multiset<double>& ms, double a, double b) {

auto it = ms.begin();

while (it != ms.end()) {

if (\*it >= a && \*it <= b) {

it = ms.erase(it);

} else {

++it;

}

}

}

void dob(multiset<double>& ms) {

if (ms.empty()) return;

double min\_val = \*ms.begin();

double max\_val = \*ms.rbegin();

double sum = min\_val + max\_val;

multiset<double> new\_ms;

for (auto x : ms) {

new\_ms.insert(x + sum);

}

ms = new\_ms;

}

int main() {

int n;

cout << "Размер multiset = ";

cin >> n;

multiset<double> ms = make\_multiset(n);

print\_multiset(ms);

DobSrAr(ms);

print\_multiset(ms);

double a, b;

cout << "Введите a: "; cin >> a;

cout << "Введите b: "; cin >> b;

remove\_key(ms, a, b);

print\_multiset(ms);

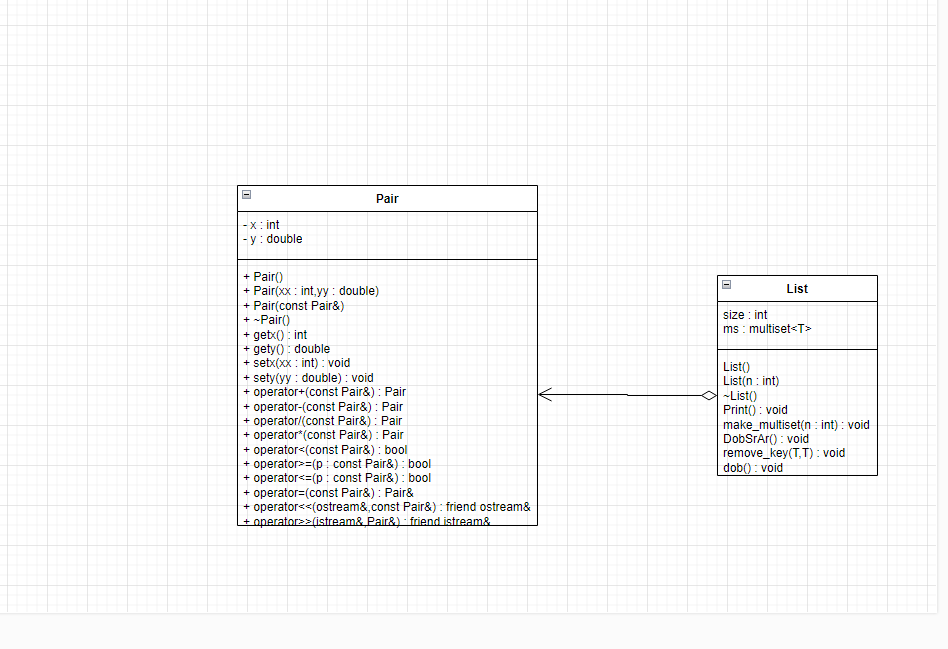
dob(ms);

print\_multiset(ms);

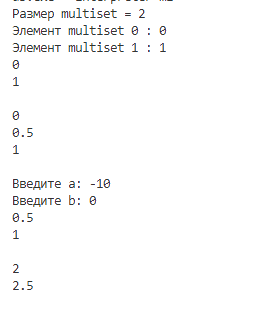
return 0;

}

UML диаграмма:



Входные и выходные данные:



Ответы на вопросы:

1. Ассоциативный контейнер — это тип контейнера данных, который хранит элементы в отсортированном порядке, где каждый элемент связан с ключом. В отличие от последовательных контейнеров, где элементы хранятся в определенном порядке, определяемом порядком добавления, в ассоциативных контейнерах порядок элементов определяется ключами. Это позволяет эффективно выполнять поиск, вставку и удаление элементов на основе ключей.
2. set: Хранит уникальные элементы в отсортированном порядке. multiset: Хранит элементы в отсортированном порядке, разрешая дубликаты. map: Хранит пары ключ-значение, где ключи уникальны и элементы отсортированы по ключам. multimap: Хранит пары ключ-значение, разрешая дубликаты ключей и элементы отсортированы по ключам. unordered\_set: Хранит уникальные элементы в произвольном порядке (на основе хеш-функции). unordered\_multiset: Хранит элементы в произвольном порядке, разрешая дубликаты (на основе хеш-функции). unordered\_map: Хранит пары ключ-значение, где ключи уникальны и элементы хранятся в произвольном порядке (на основе хеш-функции). unordered\_multimap: Хранит пары ключ-значение, разрешая дубликаты ключей и элементы хранятся в произвольном порядке (на основе хеш-функции).
3. Оператор []: Предоставляет прямой доступ к значению по ключу. Если ключ не существует, он будет вставлен в контейнер с дефолтным значением. Метод at(): Аналогичен оператору [], но генерирует исключение std::out\_of\_range, если ключ не существует. Метод find(): Возвращает итератор на элемент с указанным ключом, или end(), если ключ не найден. Эти контейнеры хранят только ключи, поэтому доступ к элементам осуществляется через итераторы. Метод find(): Возвращает итератор на элемент, если он существует, или end(), если не найден. Метод find(): Возвращает итератор на \*первый\* элемент с указанным ключом. Метод equal\_range(): Возвращает пару итераторов, определяющих диапазон элементов с указанным ключом.
4. insert(): Вставляет новый элемент (ключ или пару ключ-значение) в контейнер. erase(): Удаляет элемент (или элементы) из контейнера по ключу или итератору. Ищет элемент по ключу и возвращает итератор на него. count(): Возвращает количество элементов с указанным ключом (для multimap, multiset, unordered\_multimap, unordered\_multiset). size(): Возвращает количество элементов в контейнере. empty(): Возвращает true, если контейнер пуст, и false в противном случае. begin(): Возвращает итератор на начало контейнера. end(): Возвращает итератор на конец контейнера. clear(): Удаляет все элементы из контейнера. lower\_bound(): (только для отсортированных контейнеров set, multiset, map, multimap) Возвращает итератор на первый элемент, не меньший чем заданный ключ. upper\_bound(): (только для отсортированных контейнеров set, multiset, map, multimap) Возвращает итератор на первый элемент, больший чем заданный ключ. equal\_range(): (только для multimap и multiset) Возвращает пару итераторов, определяющих диапазон элементов с указанным ключом.

По умолчанию:

#include <iostream>

#include <map>

int main() {

std::map<std::string, int> myMap; // Создает пустой map, ключи типа string, значения типа int

return 0;

}

С использованием списка инициализации:

#include <iostream>

#include <map>

int main() {

std::map<std::string, int> myMap = {

{"apple", 1},

{"banana", 2},

{"cherry", 3}

};

return 0;

}

Копирование другого map:

#include <iostream>

#include <map>

int main() {

std::map<std::string, int> myMap1 = {{"apple", 1}, {"banana", 2}};

std::map<std::string, int> myMap2 = myMap1; // Создает копию myMap1

return 0;

}

Перемещение содержимого другого map:

#include <iostream>

#include <map>

int main() {

std::map<std::string, int> myMap1 = {{"apple", 1}, {"banana", 2}};

std::map<std::string, int> myMap2 = std::move(myMap1); // Перемещает содержимое myMap1 в myMap2 (myMap1 становится пустым)

return 0;

}

1. Элементы в контейнере map упорядочены \*по возрастанию ключей\*. Это обеспечивается использованием дерева поиска, обычно красно-черного дерева, которое гарантирует логарифмическую сложность поиска, вставки и удаления. Чтобы изменить порядок на обратный, нужно указать свой компаратор (функтор или функция), который будет использоваться для сравнения ключей. Наиболее простой способ - использовать std::greater
2. Конструкторы и деструктор: Конструктор по умолчанию, Конструктор копирования, Конструктор перемещения, Конструктор с компаратором, Конструктор из диапазона, Деструктор. Операции доступа к элементам: operator[]: Доступ к элементу по ключу (если ключ не существует, создает новый элемент). at(): Доступ к элементу по ключу (бросает исключение, если ключ не существует). find(): Поиск элемента по ключу (возвращает итератор). Операции модификации: insert(): Вставка нового элемента (или нескольких элементов). erase(): Удаление элемента (или нескольких элементов) по ключу или итератору. clear(): Удаление всех элементов. emplace(): Вставка элемента на месте (эффективнее insert в некоторых случаях). emplace\_hint(): Вставка элемента с подсказкой о местоположении. Операции, связанные с итераторами: begin(): Возвращает итератор на начало контейнера. end(): Возвращает итератор на конец контейнера. rbegin(): Возвращает обратный итератор на начало контейнера. rend(): Возвращает обратный итератор на конец контейнера. Информационные операции: empty(): Проверка, является ли контейнер пустым. size(): Возвращает количество элементов в контейнере. max\_size(): Возвращает максимальное количество элементов, которое может вместить контейнер. Операции поиска: count(): Возвращает количество элементов с указанным ключом (всегда 0 или 1 для map). lower\_bound(): Возвращает итератор на первый элемент, не меньший чем заданный ключ. upper\_bound(): Возвращает итератор на первый элемент, больший чем заданный ключ. equal\_range(): Возвращает пару итераторов, определяющих диапазон элементов с указанным ключом. Операции, связанные с компаратором: key\_comp(): Возвращает копию объекта компаратора для ключей. value\_comp(): Возвращает объект компаратора для значений (пары ключ-значение). Прочие операции: swap(): Обмен содержимым двух контейнеров. get\_allocator(): Возвращает копию объекта аллокатора.

#include <iostream>

#include <map>

// Функция для добавления элемента в map с использованием make\_pair

void addElementToMap(std::map<std::string, int>& myMap, const std::string& key, int value) {

myMap.insert(std::make\_pair(key, value));

}

int main() {

std::map<std::string, int> myMap;

// Добавляем элементы с использованием функции addElementToMap

addElementToMap(myMap, "apple", 1);

addElementToMap(myMap, "banana", 2);

// Выводим содержимое map

for (const auto& pair : myMap) {

std::cout << pair.first << ": " << pair.second << std::endl;

}

return 0;

}

9, 10,11. #include <iostream>

#include <map>

// Функция для добавления элементов в контейнер map с помощью функции операции прямого доступа []

void addElementToMapDirectAccess(std::map<std::string, int>& myMap, const std::string& key, int value) {

myMap[key] = value; // Добавляем элемент, используя оператор []

}

// Функция для печати контейнера map с помощью итератора

void printMapWithIterator(const std::map<std::string, int>& myMap) {

std::cout << "Печать map с помощью итератора:" << std::endl;

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

std::cout << it->first << ": " << it->second << std::endl; // Выводим ключ и значение

}

std::cout << std::endl;

}

// Функция для печати контейнера map с помощью функции операции прямого доступа []

void printMapWithDirectAccess(const std::map<std::string, int>& myMap) {

std::cout << "Печать map с помощью прямого доступа:" << std::endl;

for (const auto& pair : myMap) {

std::cout << pair.first << ": " << myMap.at(pair.first) << std::endl; // Выводим ключ и значение, используя at() для доступа

}

std::cout << std::endl;

}

int main() {

std::map<std::string, int> myMap;

// Добавляем элементы, используя функцию addElementToMapDirectAccess

addElementToMapDirectAccess(myMap, "apple", 1);

addElementToMapDirectAccess(myMap, "banana", 2);

addElementToMapDirectAccess(myMap, "cherry", 3);

// Печатаем map с помощью итератора

printMapWithIterator(myMap);

// Печатаем map с помощью прямого доступа

printMapWithDirectAccess(myMap);

return 0;

}

12. map: Хранит пары ключ-значение, где ключи должны быть уникальными. Если попытаться вставить элемент с ключом, который уже существует в map, старое значение для этого ключа будет перезаписано (при использовании operator[]) или вставка будет проигнорирована (при использовании insert()). Метод insert() возвращает пару, где первый элемент - итератор на вставленный (или уже существующий) элемент, а второй элемент - bool, показывающий, была ли произведена вставка (true) или нет (false). Реализует ассоциативный массив, где каждый ключ соответствует одному значению. multimap: Хранит пары ключ-значение, где ключи могут повторяться. Это значит, что один и тот же ключ может быть связан с несколькими разными значениями. Если вставить элемент с ключом, который уже существует в multimap, новый элемент будет просто добавлен к уже существующим элементам с тем же ключом. Не предоставляет оператор [] для прямого доступа, так как неясно, какое значение нужно возвращать при наличии нескольких значений для одного ключа. Для доступа к значениям необходимо использовать итераторы.

13. set — это контейнер, который хранит уникальные элементы в отсортированном порядке. Он основан на древовидной структуре данных (обычно красно-черное дерево), что обеспечивает логарифмическую сложность для поиска, вставки и удаления элементов. Ключевое свойство set - это уникальность элементов. Попытка вставить элемент, который уже существует в set, будет проигнорирована. Метод insert() вернет пару, где первый элемент - итератор на вставленный (или уже существующий) элемент, а второй элемент - bool, показывающий, была ли произведена вставка (true) или нет (false). set используется, когда необходимо хранить набор уникальных значений и иметь возможность эффективно проверять наличие элемента, добавлять новые элементы и удалять существующие.

14. map хранит пары ключ-значение (std::pair<Key, Value>), как словарь, где каждый ключ связан с определенным значением. Ключи в map должны быть уникальными. Если попытаться добавить ключ, который уже существует, старое значение будет перезаписано (при использовании operator[]). map реализует ассоциативный массив, предоставляя быстрый доступ к значению по ключу. Элементы в map отсортированы по ключам (по умолчанию - по возрастанию). Для доступа к элементам используется оператор [], метод at(), или итераторы. set хранит только ключи (Key), то есть просто набор уникальных элементов. В set нет связанных значений. Все элементы в set должны быть уникальными. Если попытаться добавить дубликат, он будет проигнорирован. set используется для хранения и проверки наличия уникального набора элементов. Он хорошо подходит для ситуаций, когда нужно поддерживать список уникальных значений и быстро проверять, присутствует ли определенное значение в этом списке. Элементы в set отсортированы по значению (по умолчанию - по возрастанию). Доступ к элементам осуществляется через итераторы (методом find()).

15. По умолчанию (создает пустой set):

#include <iostream>

#include <set>

int main() {

std::set<int> mySet; // Создает пустой set, хранящий int

return 0;

}

С использованием списка инициализации (initializer list):

#include <iostream>

#include <set>

int main() {

std::set<int> mySet = {3, 1, 4, 1, 5, 9, 2, 6}; // Создаст set {1, 2, 3, 4, 5, 6, 9} (дубликаты игнорируются)

return 0;

}

Копирование другого set:

#include <iostream>

#include <set>

int main() {

std::set<int> mySet1 = {1, 2, 3};

std::set<int> mySet2 = mySet1; // Создаст копию mySet1

return 0;

}

Перемещение содержимого другого set:

#include <iostream>

#include <set>

int main() {

std::set<int> mySet1 = {1, 2, 3};

std::set<int> mySet2 = std::move(mySet1); // Переместит содержимое mySet1 в mySet2 (mySet1 станет пустым)

return 0;

}

Из диапазона элементов (например, из массива):

#include <iostream>

#include <set>

int main() {

int arr[] = {5, 2, 8, 1, 9, 4};

std::set<int> mySet(arr, arr + sizeof(arr) / sizeof(arr[0])); // Создаст set из элементов массива arr

return 0;

}

1. Элементы в контейнере set упорядочены по возрастанию. Это гарантируется использованием дерева поиска, которое автоматически поддерживает элементы в отсортированном порядке.
2. Чтобы изменить порядок на обратный, нужно указать свой компаратор при создании set. Самый простой способ - использовать std::greater<T>, где T - тип элементов в set:

#include <iostream>

#include <set>

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

int main() {

// Создаем set с компаратором std::greater<int>

std::set<int, std::greater<int>> mySet;

mySet.insert(5);

mySet.insert(2);

mySet.insert(8);

for (int x : mySet) {

std::cout << x << " "; // Выведет 8 5 2

}

std::cout << std::endl;

return 0;

}

18, 19.

#include <iostream>

#include <set>

// Функция для добавления элементов в контейнер set

void addElementToSet(std::set<int>& mySet, int element) {

mySet.insert(element); // Добавляем элемент в set

}

// Функция для печати контейнера set

void printSet(const std::set<int>& mySet) {

std::cout << "Содержимое set: ";

for (int element : mySet) {

std::cout << element << " ";

}

std::cout << std::endl;

}

int main() {

std::set<int> mySet;

// Добавляем элементы, используя функцию addElementToSet

addElementToSet(mySet, 5);

addElementToSet(mySet, 2);

addElementToSet(mySet, 8);

addElementToSet(mySet, 2); // Попытка добавить дубликат (будет проигнорирована)

addElementToSet(mySet, 1);

// Печатаем set с помощью функции printSet

printSet(mySet);

return 0;

}

20. set: Хранит уникальные элементы. Дубликаты не допускаются. Если вы попытаетесь вставить элемент, который уже существует в set, вставка будет проигнорирована. Метод insert() возвращает std::pair<iterator, bool>, где iterator указывает на элемент (существующий или вставленный), а bool показывает, была ли произведена вставка (true) или нет (false). multiset: Позволяет хранить дубликаты. Один и тот же элемент может встречаться в multiset несколько раз. Метод insert() возвращает только iterator на вставленный элемент.

**Лабораторная работа 13:**

Код программы:

Класс Pair:  
#include <iostream>

#include <sstream>

using namespace std;

class Pair{

int x;

double y;

public:

Pair();

Pair(int xx,double yy);

Pair(const Pair&);

~Pair();

int getx();

double gety();

void setx(int xx);

void sety(double yy);

Pair operator+(const Pair&) const;

Pair operator-(const Pair&) const;

Pair operator/(const Pair&) const;

Pair operator\*(const Pair&) const;

bool operator<(const Pair&) const;

bool operator>(const Pair&) const;

bool operator>=(const Pair& p) const;

bool operator<=(const Pair& p) const;

Pair& operator=(const Pair&);

friend ostream& operator<<(ostream&,const Pair&);

friend istream& operator>>(istream&,Pair&);

};

Pair::Pair(){

x=0;

y=0;

}

Pair::Pair(int xx,double yy){

x=xx;

y=yy;

}

Pair::Pair(const Pair& p){

x=p.x;

y=p.y;

}

Pair::~Pair(){

}

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

x=p.x;

y=p.y;

return \*this;

}

int Pair::getx(){

return x;

}

double Pair::gety(){

return y;

}

void Pair::setx(int xx){

x=xx;

}

void Pair::sety(double yy){

y=yy;

}

Pair Pair::operator+(const Pair& p) const

{

return Pair(x+p.x,y+p.y);

}

Pair Pair::operator-(const Pair& p) const

{

return Pair(x-p.x,y-p.y);

}

Pair Pair::operator\*(const Pair& p) const

{

return Pair(x\*p.x,y\*p.y);

}

Pair Pair::operator/(const Pair& p) const

{

return Pair(x/p.x,y/p.y);

}

istream& operator>>(istream& fin,Pair& p){

cout<<"x= ";

fin>>p.x;

cout<<"y= ";

fin>>p.y;

return fin;

}

ostream& operator<<(ostream& out,const Pair& p){

out<<'('<<p.x<<":"<<p.y<<")";

return out;

}

bool Pair::operator>=(const Pair& p) const {

return (x > p.x) || (x==p.x)&& (y >= p.y);

}

bool Pair::operator<=(const Pair& p) const {

return (x < p.x) || (x==p.x)&& (y <= p.y);

}

bool Pair::operator<(const Pair& p) const{

return (x < p.x) || (x==p.x)&& (y < p.y);

}

bool Pair::operator>(const Pair& p) const{

return (x > p.x) || (x==p.x)&& (y > p.y);

}

Функции main:

#include <list>

#include <iostream>

#include "pair.h"

#include <algorithm>

using namespace std;

list<Pair> make\_list(int n){

list<Pair> lst;

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

cout<<"Элемент списка "<<i<<" : ";

Pair h;

cin>>h;

lst.push\_back(h);

}

return lst;

}

void print\_list(list<Pair>& lst){

for(const Pair& x:lst){

cout<<x<<endl;

}

cout<<endl;

}

void DobSrAr(list<Pair>& lst){

Pair sum;

for\_each(lst.begin(),lst.end(),[&sum](Pair& p){sum=sum+p;});

sum.setx(sum.getx()/lst.size());

sum.sety(sum.gety()/lst.size());

lst.push\_back(sum);

}

void remove\_key(list<Pair>& lst,Pair a,Pair b){

list<Pair>::iterator it=remove\_if(lst.begin(),lst.end(),[a,b](Pair& p){return p>=a && p<=b;});

lst.erase(it,lst.end());

}

void dob(list<Pair>& lst){

list<Pair>::iterator it=min\_element(lst.begin(),lst.end());

list<Pair>::iterator it1=max\_element(lst.begin(),lst.end());

Pair sum=(\*it)+(\*it1);

for\_each(lst.begin(),lst.end(),[sum](Pair& p){p=p+sum;});

}

int main(){

int n;

cout<<"Размер списка = ";

cin>>n;

list<Pair> lst;

lst=make\_list(n);

print\_list(lst);

DobSrAr(lst);

print\_list(lst);

Pair a,b;

cout<<"Введите a: "; cin>>a;

cout<<"Введите b: "; cin>>b;

remove\_key(lst,a,b);

print\_list(lst);

dob(lst);

print\_list(lst);

return 0;

}

#include <queue>

#include <vector>

#include <iostream>

#include "pair.h"

using namespace std;

void print\_pqueue(priority\_queue<Pair>& pq) {

priority\_queue<Pair> temp = pq;

while (!temp.empty()) {

cout << temp.top() << endl;

temp.pop();

}

cout << endl;

}

void DobSrAr(priority\_queue<Pair>& pq) {

if (pq.empty()) return;

Pair sum;

int kol = pq.size();

priority\_queue<Pair> temp = pq;

while (!temp.empty()) {

sum =sum+ temp.top();

temp.pop();

}

sum.setx(sum.getx()/kol);

sum.sety(sum.gety()/kol);

pq.push(sum);

}

void remove\_key(priority\_queue<Pair>& pq, Pair a, Pair b) {

priority\_queue<Pair> new\_pq;

while (!pq.empty()) {

Pair val = pq.top();

pq.pop();

if (val < a || val > b) {

new\_pq.push(val);

}

}

pq = new\_pq;

}

void dob(priority\_queue<Pair>& pq) {

if (pq.empty()) return;

priority\_queue<Pair> temp = pq;

Pair max\_val = temp.top();

Pair min\_val = max\_val;

while (!temp.empty()) {

min\_val = temp.top();

temp.pop();

}

Pair sum = max\_val + min\_val;

priority\_queue<Pair> new\_pq;

while (!pq.empty()) {

new\_pq.push(pq.top() + sum);

pq.pop();

}

pq = new\_pq;

}

int main() {

int n;

cout << "Размер очереди = ";

cin >> n;

priority\_queue<Pair> pq;

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

cout << "Элемент " << i << " : ";

Pair h;

cin >> h;

pq.push(h);

}

print\_pqueue(pq);

DobSrAr(pq);

print\_pqueue(pq);

Pair a, b;

cout << "Введите a: "; cin >> a;

cout << "Введите b: "; cin >> b;

remove\_key(pq, a, b);

print\_pqueue(pq);

dob(pq);

print\_pqueue(pq);

return 0;

}

#include <queue>

#include <vector>

#include <iostream>

#include "pair.h"

#include <map>

using namespace std;

void print\_m(map<int,Pair>& m) {

map<int,Pair> temp = m;

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

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

}

cout << endl;

}

void DobSrAr(map<int,Pair>& m) {

if (m.empty()) return;

Pair sum;

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

sum=sum+it->second;

}

sum.setx(sum.getx()/m.size());

sum.sety(sum.gety()/m.size());

m.insert(make\_pair(m.size(),sum));

}

void remove\_key(map<int,Pair>& m, Pair a, Pair b) {

map<int,Pair> m1;

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

if (it->second < a ||it->second> b) {

m1.insert(\*it);

}

}

m = m1;

}

void dob(map<int,Pair>& m) {

Pair min=m.begin()->second;

Pair max=m.begin()->second;

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

if (it->second >max) {

max=it->second;

}

if (it->second <min) {

min=it->second;

}

}

map<int,Pair> m1;

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

m1.insert(make\_pair(it->first,it->second+max+min));

}

m = m1;

}

int main() {

int n;

cout << "Размер словаря = ";

cin >> n;

map<int,Pair> m;

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

cout << "Элемент " << i << " : ";

Pair h;

cin >> h;

m.insert(make\_pair(i,h));

}

print\_m(m);

DobSrAr(m);

print\_m(m);

Pair a, b;

cout << "Введите a: "; cin >> a;

cout << "Введите b: "; cin >> b;

remove\_key(m, a, b);

print\_m(m);

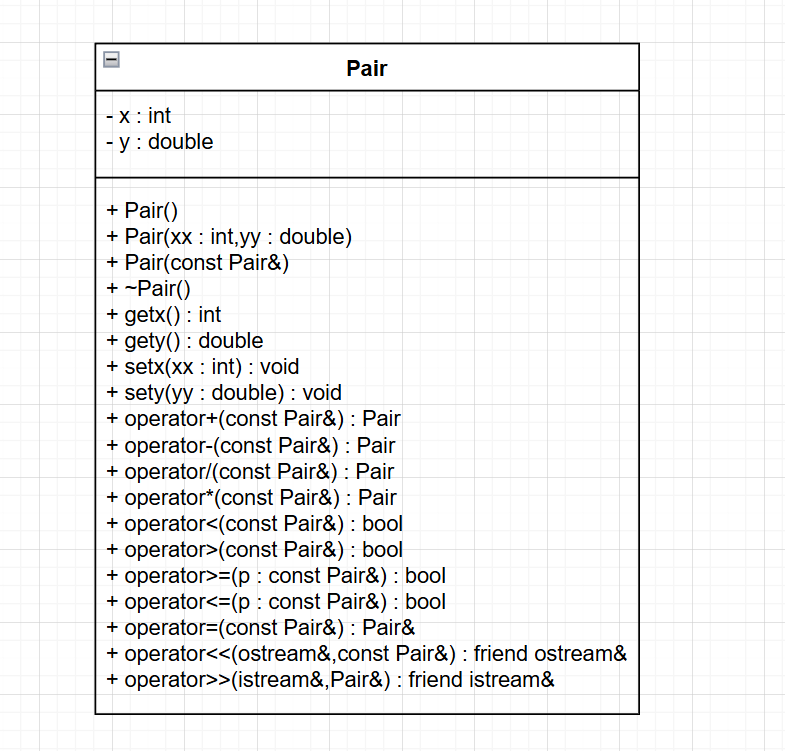
dob(m);

print\_m(m);

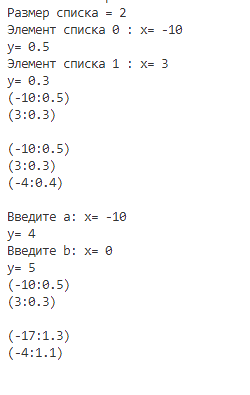
return 0;

}

UML диаграмма:



Входные и выходные данные:



https://github.com/Prefix008/lab.git