1. Реализуйте АТД Стек на односвязном списке (forward\_list).

#include <iostream>

#include <forward\_list>

using namespace std;

template <class T>

class Stack {

private:

forward\_list<T> data;

public:

void push(const T& value) {

data.push\_front(value);

}

void pop() {

if (!data.empty()) {

data.pop\_front();

} else {

cout << "Стек пустой." << endl;

}

}

T top() const {

if (!data.empty()) {

return data.front();

} else {

cout << "Стек пустой." << endl;

return T();

}

}

bool isEmpty() const {

return data.empty();

}

};

5. Напишите функцию для симметричного (in-order) обхода бинарного дерева, заданного следующей структурой:

struct node { int value; node \*left, \*right; };

К каждому значению, хранящемуся в дереве, функция применяет функцию, указанную в качестве аргумента:

void inorder(node \*n, void (\*f)(int));

void inorder(node \*n, void (\*f)(int)) {

if (n == nullptr) {

return;

}

inorder(n->left, f); // Рекурсивно обходим левое поддерево

f(n->value); // Применяем функцию к текущему узлу

inorder(n->right, f); // Рекурсивно обходим правое поддерево

}

10. Определить АТД Полином, обеспечивающий метод calc для вычисления значения полинома в точке x. Реализовать полином через представление в виде вектора коэффициентов. В конструкторе задается набор коэффициентов a0,a1,...,an−1 . Определить операции + и \*.

#include <iostream>

#include <vector>

using namespace std;

class Polynomial {

public:

// Конструктор, принимающий вектор коэффициентов

Polynomial(const vector<double>& coefficients) : coeffs(coefficients) {}

// Вычисление значения полинома в точке x

double calc(double x) const {

double result = 0;

for (int i = coeffs.size() - 1; i >= 0; --i) {

result = result \* x + coeffs[i];

}

return result;

}

// Оператор сложения

Polynomial operator+(const Polynomial& other) const {

vector<double> resultCoeffs(max(coeffs.size(), other.coeffs.size()));

for (size\_t i = 0; i < resultCoeffs.size(); ++i) {

resultCoeffs[i] = (i < coeffs.size() ? coeffs[i] : 0) + (i < other.coeffs.size() ? other.coeffs[i] : 0);

}

return Polynomial(resultCoeffs);

}

// Оператор умножения

Polynomial operator\*(const Polynomial& other) const {

vector<double> resultCoeffs(coeffs.size() + other.coeffs.size() - 1);

for (size\_t i = 0; i < coeffs.size(); ++i) {

for (size\_t j = 0; j < other.coeffs.size(); ++j) {

resultCoeffs[i + j] += coeffs[i] \* other.coeffs[j];

}

}

return Polynomial(resultCoeffs);

}

private:

vector<double> coeffs; // Вектор коэффициентов

};

int main() {

// Примеры использования

Polynomial p1({1, 2, 3}); // 1 + 2x + 3x^2

Polynomial p2({4, 5}); // 4 + 5x

cout << "p1(2) = " << p1.calc(2) << endl; // 17

Polynomial p3 = p1 + p2; // 5 + 7x + 3x^2

cout << "p3(1) = " << p3.calc(1) << endl; // 15

Polynomial p4 = p1 \* p2; // 4 + 13x + 22x^2 + 15x^3

cout << "p4(0) = " << p4.calc(0) << endl; // 4

return 0;

}

13. Используя поиск в глубину, определите число компонент связности в графе, задаваемом следующим образом:

Как матрица N×M из символов латинского алфавита. Клетки считаются связными, если в них находится одинаковая буква и они имеют общую границу.

#include <iostream>

#include <vector>

using namespace std;

// Функция для выполнения поиска в глубину (DFS)

void dfs(const vector<vector<char>>& graph, vector<vector<bool>>& visited, int i, int j) {

int n = graph.size();

int m = graph[0].size();

visited[i][j] = true; // Помечаем текущую клетку как посещенную

// Проверяем соседние клетки (вверх, вниз, влево, вправо)

if (i > 0 && graph[i - 1][j] == graph[i][j] && !visited[i - 1][j]) {

dfs(graph, visited, i - 1, j);

}

if (i < n - 1 && graph[i + 1][j] == graph[i][j] && !visited[i + 1][j]) {

dfs(graph, visited, i + 1, j);

}

if (j > 0 && graph[i][j - 1] == graph[i][j] && !visited[i][j - 1]) {

dfs(graph, visited, i, j - 1);

}

if (j < m - 1 && graph[i][j + 1] == graph[i][j] && !visited[i][j + 1]) {

dfs(graph, visited, i, j + 1);

}

}

// Функция для определения числа компонент связности

int countComponents(const vector<vector<char>>& graph) {

int n = graph.size();

int m = graph[0].size();

vector<vector<bool>> visited(n, vector<bool>(m, false)); // Матрица для отслеживания посещенных клеток

int components = 0; // Счетчик компонент связности

// Проходим по всем клеткам матрицы

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

for (int j = 0; j < m; ++j) {

if (!visited[i][j]) { // Если клетка не посещена

dfs(graph, visited, i, j); // Запускаем DFS из этой клетки

components++; // Увеличиваем счетчик компонент

}

}

}

return components;

}

int main() {

int n, m;

cout << "Введите размеры матрицы (N x M): ";

cin >> n >> m;

vector<vector<char>> graph(n, vector<char>(m));

cout << "Введите матрицу графа:" << endl;

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

for (int j = 0; j < m; ++j) {

cin >> graph[i][j];

}

}

int components = countComponents(graph);

cout << "Число компонент связности: " << components << endl;

return 0;

}

23. Определите необходимые геометрические объекты и напишите следующую функцию  
В декартовой системе координат на плоскости заданы координаты концов отрезка и ещё одной точки. Определить расстояние от точки до отрезка.

#include <iostream>

#include <cmath>

using namespace std;

// Структура для представления точки на плоскости

struct Point {

double x;

double y;

};

// Функция для вычисления расстояния между двумя точками

double distance(Point A, Point B) {

return sqrt(pow(A.x - B.x, 2) + pow(A.y - B.y, 2));

}

// Функция для вычисления площади треугольника по формуле Герона

double triangleArea(double a, double b, double c) {

double s = (a + b + c) / 2;

return sqrt(s \* (s - a) \* (s - b) \* (s - c));

}

// Функция для вычисления расстояния от точки до отрезка

double distanceToSegment(Point A, Point B, Point P) {

// Длины сторон треугольника ABP

double a = distance(A, B);

double b = distance(B, P);

double c = distance(A, P);

// Площадь треугольника ABP

double S = triangleArea(a, b, c);

// Высота треугольника ABP, проведенная из точки P

double h = 2 \* S / a;

// Проверка, лежит ли проекция точки P на отрезок AB

if (a \* a + b \* b >= c \* c && a \* a + c \* c >= b \* b) {

return h; // Расстояние равно высоте

} else {

return min(b, c); // Расстояние равно расстоянию до ближайшего конца отрезка

}

}

int main() {

// Координаты концов отрезка

Point A, B;

cout << "Введите координаты точки A (x y): ";

cin >> A.x >> A.y;

cout << "Введите координаты точки B (x y): ";

cin >> B.x >> B.y;

// Координаты точки

Point P;

cout << "Введите координаты точки P (x y): ";

cin >> P.x >> P.y;

// Вычисление расстояния

double dist = distanceToSegment(A, B, P);

cout << "Расстояние от точки до отрезка: " << dist << endl;

return 0;

}

25. Напишите функцию разложения числа на простые множители.

#include <iostream>

#include <vector>

using namespace std;

// Функция для разложения числа на простые множители

vector<int> primeFactorization(int n) {

vector<int> factors;

// Обработка случая, когда n <= 1

if (n <= 1) {

return factors;

}

// Деление на 2, пока число делится на 2

while (n % 2 == 0) {

factors.push\_back(2);

n /= 2;

}

// Проверка делимости на нечетные числа, начиная с 3

for (int i = 3; i \* i <= n; i += 2) {

while (n % i == 0) {

factors.push\_back(i);

n /= i;

}

}

// Если n > 1, значит, n - простое число

if (n > 1) {

factors.push\_back(n);

}

return factors;

}

int main() {

int n;

cout << "Введите число: ";

cin >> n;

// Разложение числа на простые множители

vector<int> factors = primeFactorization(n);

// Вывод результатов

cout << "Простые множители числа " << n << ": ";

for (int factor : factors) {

cout << factor << " ";

}

cout << endl;

return 0;

}