# **Однозв’язні списки**

struct Node {

int data;

Node\* next;

};

### **1. Вилучити всі елементи з від’ємними значеннями**

void deleteNegatives(Node\*& head) {

while (head && head->data < 0) {

Node\* tmp = head;

head = head->next;

delete tmp;

}

Node\* cur = head;

while (cur && cur->next) {

if (cur->next->data < 0) {

Node\* tmp = cur->next;

cur->next = cur->next->next;

delete tmp;

} else {

cur = cur->next;

}

}

}

### **2. Копіювання списку з оберненим порядком**

Node\* copyReversed(Node\* head) {

Node\* newHead = NULL;

Node\* cur = head;

while (cur) {

Node\* n = new Node;

n->data = cur->data;

n->next = newHead;

newHead = n;

cur = cur->next;

}

return newHead;

}

### **3. Перемістити найбільший елемент у кінець**

void moveMaxToEnd(Node\*& head) {

if (!head || !head->next) return;

Node\* maxPrev = NULL, \*maxNode = head;

Node\* prev = head, \*cur = head->next;

while (cur) {

if (cur->data > maxNode->data) {

maxPrev = prev;

maxNode = cur;

}

prev = cur;

cur = cur->next;

}

if (!maxNode->next) return;

if (maxPrev) maxPrev->next = maxNode->next;

else head = head->next;

prev->next = maxNode;

maxNode->next = NULL;

}

### **4. Вилучити вузли у парних позиціях**

void deleteEvenPositions(Node\*& head) {

if (!head) return;

Node\* cur = head;

while (cur && cur->next) {

Node\* tmp = cur->next;

cur->next = cur->next->next;

delete tmp;

cur = cur->next;

}

}

### **5. Впорядкувати: непарні позиції → парні позиції**

Node\* reorderOddEven(Node\* head) {

if (!head || !head->next) return head;

Node\* odd = head, \*even = head->next;

Node\* evenHead = even;

while (even && even->next) {

odd->next = even->next;

odd = odd->next;

even->next = odd->next;

even = even->next;

}

odd->next = evenHead;

return head;

}

### **6. Впорядкувати за зростанням («сортування вибором»)**

void selectionSort(Node\* head) {

for (Node\* i = head; i; i = i->next) {

Node\* minNode = i;

for (Node\* j = i->next; j; j = j->next)

if (j->data < minNode->data) minNode = j;

int tmp = i->data;

i->data = minNode->data;

minNode->data = tmp;

}

}

### **7. Впорядкувати за спаданням («сортування вставкою»)**

Node\* insertionSortDesc(Node\* head) {

Node\* sorted = NULL;

while (head) {

Node\* cur = head;

head = head->next;

if (!sorted || cur->data > sorted->data) {

cur->next = sorted;

sorted = cur;

} else {

Node\* p = sorted;

while (p->next && p->next->data > cur->data) p = p->next;

cur->next = p->next;

p->next = cur;

}

}

return sorted;

}

# **Двозв’язні списки**

struct DNode {

int data;

DNode\* next;

DNode\* prev;

};

### **1. Впорядкувати у спадному порядку**

void sortDesc(DNode\*& head) {

for (DNode\* i = head; i; i = i->next) {

DNode\* maxNode = i;

for (DNode\* j = i->next; j; j = j->next)

if (j->data > maxNode->data) maxNode = j;

int tmp = i->data;

i->data = maxNode->data;

maxNode->data = tmp;

}

}

### **2. Копіювати однозв’язний → зробити двозв’язний циклічний**

DNode\* copyToDoublyCircular(Node\* head) {

if (!head) return NULL;

DNode\* newHead = new DNode{head->data, NULL, NULL};

DNode\* last = newHead;

Node\* cur = head->next;

while (cur) {

DNode\* n = new DNode{cur->data, NULL, last};

last->next = n;

last = n;

cur = cur->next;

}

newHead->prev = last;

last->next = newHead;

return newHead;

}

# **Циклічні списки**

struct CNode {

int data;

CNode\* next;

};

### **1. Відсортувати: від’ємні → 0 → додатні**

void partitionList(CNode\*& head) {

if (!head) return;

CNode \*neg=NULL, \*zero=NULL, \*pos=NULL;

CNode \*nLast=NULL, \*zLast=NULL, \*pLast=NULL;

CNode\* cur = head;

do {

if (cur->data < 0) {

if (!neg) neg = nLast = cur;

else { nLast->next = cur; nLast = cur; }

} else if (cur->data == 0) {

if (!zero) zero = zLast = cur;

else { zLast->next = cur; zLast = cur; }

} else {

if (!pos) pos = pLast = cur;

else { pLast->next = cur; pLast = cur; }

}

cur = cur->next;

} while (cur != head);

if (nLast) nLast->next = zero ? zero : pos ? pos : neg;

if (zLast) zLast->next = pos ? pos : neg;

if (pLast) pLast->next = neg ? neg : zero;

head = neg ? neg : (zero ? zero : pos);

}

### **2. Розділити список на 2: додатні та інші**

void splitList(CNode\* head, CNode\*& posList, CNode\*& otherList) {

if (!head) return;

posList = otherList = NULL;

CNode \*pLast=NULL, \*oLast=NULL;

CNode\* cur = head;

do {

if (cur->data > 0) {

if (!posList) posList = pLast = cur;

else { pLast->next = cur; pLast = cur; }

} else {

if (!otherList) otherList = oLast = cur;

else { oLast->next = cur; oLast = cur; }

}

cur = cur->next;

} while (cur != head);

if (pLast) pLast->next = posList;

if (oLast) oLast->next = otherList;

}

### **3. Вилучити всі елементи з максимальним значенням**

void deleteMax(CNode\*& head) {

if (!head) return;

int maxVal = head->data;

CNode\* cur = head->next;

while (cur != head) {

if (cur->data > maxVal) maxVal = cur->data;

cur = cur->next;

}

while (head && head->data == maxVal) {

if (head->next == head) { delete head; head = NULL; return; }

CNode\* tmp = head;

CNode\* last = head;

while (last->next != head) last = last->next;

head = head->next;

last->next = head;

delete tmp;

}

cur = head;

while (cur && cur->next != head) {

if (cur->next->data == maxVal) {

CNode\* tmp = cur->next;

cur->next = tmp->next;

delete tmp;

} else cur = cur->next;

}

}

# **Бінарні дерева пошуку (BST)**

struct Node {

int data;

Node\* left;

Node\* right;

};

### **1. Порахувати кількість вершин більших за v**

int countGreater(Node\* root, int v) {

if (!root) return 0;

int count = (root->data > v) ? 1 : 0;

return count + countGreater(root->left, v) + countGreater(root->right, v);

}

### **2. Порахувати кількість вершин менших за v**

int countLess(Node\* root, int v) {

if (!root) return 0;

int count = (root->data < v) ? 1 : 0;

return count + countLess(root->left, v) + countLess(root->right, v);

}

### **3. Сума значень вершин більших за v**

int sumGreater(Node\* root, int v) {

if (!root) return 0;

int sum = (root->data > v) ? root->data : 0;

return sum + sumGreater(root->left, v) + sumGreater(root->right, v);

}

### **4. Сума значень вершин менших за v**

int sumLess(Node\* root, int v) {

if (!root) return 0;

int sum = (root->data < v) ? root->data : 0;

return sum + sumLess(root->left, v) + sumLess(root->right, v);

}

### **5. Знайти найменше значення > v**

int minGreater(Node\* root, int v, int ans = -1) {

if (!root) return ans;

if (root->data > v)

return minGreater(root->left, v, root->data);

return minGreater(root->right, v, ans);

}

### **6. Вилучити найбільший і найменший елемент**

Node\* deleteMin(Node\* root) {

if (!root) return NULL;

if (!root->left) {

Node\* r = root->right;

delete root;

return r;

}

root->left = deleteMin(root->left);

return root;

}

Node\* deleteMax(Node\* root) {

if (!root) return NULL;

if (!root->right) {

Node\* l = root->left;

delete root;

return l;

}

root->right = deleteMax(root->right);

return root;

}

### **7. Розбити дерево на <K та ≥K**

void splitBST(Node\* root, int K, Node\*& less, Node\*& greater) {

if (!root) { less = greater = NULL; return; }

if (root->data < K) {

splitBST(root->right, K, root->right, greater);

less = root;

} else {

splitBST(root->left, K, less, root->left);

greater = root;

}

}

### **8. Об’єднати два дерева (<K та ≥K)**

Node\* joinBST(Node\* less, Node\* greater) {

if (!less) return greater;

if (!greater) return less;

Node\* cur = less;

while (cur->right) cur = cur->right;

cur->right = greater;

return less;

}

# **Звичайні бінарні дерева**

(не впорядковані, "standard form")

### **1. Порахувати кількість вершин із значенням v**

int countValue(Node\* root, int v) {

if (!root) return 0;

int count = (root->data == v) ? 1 : 0;

return count + countValue(root->left, v) + countValue(root->right, v);

}

### **2. Порахувати кількість листів**

int countLeaves(Node\* root) {

if (!root) return 0;

if (!root->left && !root->right) return 1;

return countLeaves(root->left) + countLeaves(root->right);

}

### **3. Вилучити всі листя зі значенням v**

Node\* deleteLeaves(Node\* root, int v) {

if (!root) return NULL;

if (!root->left && !root->right && root->data == v) {

delete root;

return NULL;

}

root->left = deleteLeaves(root->left, v);

root->right = deleteLeaves(root->right, v);

return root;

}

### **4. Знайти висоту дерева степені 3**

int height3(Node\* root) {

if (!root) return 0;

int h1 = height3(root->left);

int h2 = height3(root->right);

// якщо треба ще "третю дитину", треба додати поле в Node!

return 1 + (h1 > h2 ? h1 : h2);

}

### **5. Знайти кількість вершин на шляху до v**

int pathLength(Node\* root, int v) {

if (!root) return 0;

if (root->data == v) return 1;

int left = pathLength(root->left, v);

int right = pathLength(root->right, v);

if (left) return left + 1;

if (right) return right + 1;

return 0;

}

# **Дерева Базова структура вузла для бінарного дерева**

struct Node {

int val;

Node\* left;

Node\* right;

};

## **1. Вставка у бінарне дерево пошуку (BST)**

Node\* insert(Node\* root, int x) {

if (!root) {

Node\* node = new Node{x, NULL, NULL};

return node;

}

if (x < root->val) root->left = insert(root->left, x);

else root->right = insert(root->right, x);

return root;

}

## **2. Пошук елемента в BST**

bool search(Node\* root, int x) {

if (!root) return false;

if (root->val == x) return true;

if (x < root->val) return search(root->left, x);

return search(root->right, x);

}

## **5. Знайти найменший елемент більший за v**

(типово у білетах як "найменше значення, більше за v")

int minGreater(Node\* root, int v, int ans=1e9) {

if (!root) return ans;

if (root->val > v)

return minGreater(root->left, v, root->val);

return minGreater(root->right, v, ans);

}

## **7. Висота дерева (звичайного бінарного або n-арного)**

int height(Node\* root) {

if (!root) return 0;

return 1 + max(height(root->left), height(root->right));

}

## **8. Обходи дерев**

* **Прямий (NLR)**:

void preorder(Node\* root) {

if (!root) return;

cout << root->val << " ";

preorder(root->left);

preorder(root->right);

}

* **Симетричний (LNR)**:

void inorder(Node\* root) {

if (!root) return;

inorder(root->left);

cout << root->val << " ";

inorder(root->right);

}

* **Зворотній (LRN)**:

void postorder(Node\* root) {

if (!root) return;

postorder(root->left);

postorder(root->right);

cout << root->val << " ";

}

* **Рівневий порядок (через чергу)**:

#include <queue>

void levelOrder(Node\* root) {

if (!root) return;

queue<Node\*> q;

q.push(root);

while(!q.empty()) {

Node\* cur = q.front(); q.pop();

cout << cur->val << " ";

if (cur->left) q.push(cur->left);

if (cur->right) q.push(cur->right);

}

}

## **10. AVL-дерево (тільки базова вставка зі збалансуванням)**

int height(Node\* n) { return n ? max(height(n->left), height(n->right))+1 : 0; }

int balance(Node\* n) { return n ? height(n->left)-height(n->right) : 0; }

Node\* rotateRight(Node\* y) {

Node\* x=y->left;

Node\* T=x->right;

x->right=y;

y->left=T;

return x;

}

Node\* rotateLeft(Node\* x) {

Node\* y=x->right;

Node\* T=y->left;

y->left=x;

x->right=T;

return y;

}

Node\* insertAVL(Node\* root, int key) {

if (!root) return new Node{key,NULL,NULL};

if (key<root->val) root->left=insertAVL(root->left,key);

else if (key>root->val) root->right=insertAVL(root->right,key);

else return root;

int bf=balance(root);

if (bf>1 && key<root->left->val) return rotateRight(root);

if (bf<-1 && key>root->right->val) return rotateLeft(root);

if (bf>1 && key>root->left->val) {

root->left=rotateLeft(root->left);

return rotateRight(root);

}

if (bf<-1 && key<root->right->val) {

root->right=rotateRight(root->right);

return rotateLeft(root);

}

return root;

}

## **11. Копіювання дерева**

Node\* copyTree(Node\* root) {

if (!root) return NULL;

Node\* node = new Node{root->val, NULL, NULL};

node->left = copyTree(root->left);

node->right = copyTree(root->right);

return node;

}

# **Графи (матриця суміжності)**

const int N = 100; // максимум вершин

int adj[N][N]; // матриця суміжності

### **1. Перевірка зв’язності (BFS)**

void bfs(int n, int start, bool visited[]) {

for (int i=0;i<n;i++) visited[i]=false;

int q[N], front=0, back=0;

q[back++] = start;

visited[start] = true;

while (front<back) {

int u = q[front++];

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

if (adj[u][v] && !visited[v]) {

visited[v] = true;

q[back++] = v;

}

}

}

}

### **2. Кількість компонент зв’язності**

int countComponents(int n) {

bool visited[N];

int cnt = 0;

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

if (!visited[i]) {

bfs(n,i,visited);

cnt++;

}

}

return cnt;

}

### **3. Перевірка Ейлеровості**

(граф зв’язний + усі вершини парного степеня)

bool isEulerian(int n) {

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

int deg=0;

for (int j=0;j<n;j++) deg+=adj[i][j];

if (deg%2!=0) return false;

}

return true; // треба ще перевірити зв’язність!

}

### **4. Побудова кістякового дерева (BFS)**

void spanningTree(int n, int start) {

bool visited[N]={0};

int q[N], front=0, back=0;

q[back++] = start;

visited[start] = true;

while (front<back) {

int u=q[front++];

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

if (adj[u][v] && !visited[v]) {

cout << u << " - " << v << endl; // ребро в дерево

visited[v]=true;

q[back++] = v;

}

}

}

}

### **5. Вилучення ребра (u,v)**

void removeEdge(int u, int v) {

adj[u][v]=0;

adj[v][u]=0;

}

### **6. Додавання дуги (орієнтований граф)**

void addArc(int u, int v) {

adj[u][v]=1;

}

### **7. Вилучення петель**

void removeLoops(int n) {

for (int i=0; i<n; i++) adj[i][i]=0;

}

### **8. Вилучення кратних ребер (мультіграф)**

void removeMultipleEdges(int n) {

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

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

if (adj[i][j]>1) adj[i][j]=1;

}

### **9. Вилучення ізольованих вершин**

bool isIsolated(int n, int v) {

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

if (adj[v][i] || adj[i][v]) return false;

return true;

}

# **Графи (список суміжності)**

struct Node {

int v;

Node\* next;

};

Node\* adjList[N]; // масив списків

### **1. Перевірка суміжності двох вершин**

bool areAdjacent(int u, int v) {

Node\* cur = adjList[u];

while (cur) {

if (cur->v==v) return true;

cur=cur->next;

}

return false;

}

### **2. Побудова списку суміжності з матриці**

void buildFromMatrix(int n) {

for (int i=0;i<n;i++) adjList[i]=NULL;

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

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

if (adj[i][j]) {

Node\* newNode=new Node{j, adjList[i]};

adjList[i]=newNode;

}

}

### **3. Побудова зі списку ребер**

void addEdge(int u, int v) {

Node\* n1=new Node{v, adjList[u]};

adjList[u]=n1;

Node\* n2=new Node{u, adjList[v]};

adjList[v]=n2;

}

### **4. Кількість вершин степеня 1 / 2 / 3**

int countDegree(Node\* adjList[], int n, int k) {

int cnt=0;

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

int deg=0;

Node\* cur=adjList[i];

while(cur){deg++; cur=cur->next;}

if (deg==k) cnt++;

}

return cnt;

}

## **1. DFS (глибина) і BFS (ширина)**

void dfs(int v, vector<int>& visited, vector<vector<int>>& adj) {

visited[v] = 1;

cout << v << " ";

for (int u : adj[v]) if (!visited[u]) dfs(u, visited, adj);

}

void bfs(int start, vector<vector<int>>& adj) {

vector<int> visited(adj.size(), 0);

queue<int> q;

visited[start] = 1;

q.push(start);

while(!q.empty()) {

int v = q.front(); q.pop();

cout << v << " ";

for (int u : adj[v]) if (!visited[u]) {

visited[u]=1;

q.push(u);

}

}

}

**4. Ізольовані вершини**

vector<int> isolatedVertices(vector<vector<int>>& adj) {

vector<int> iso;

for (int i=0;i<adj.size();i++) {

if (adj[i].empty()) iso.push\_back(i);

}

return iso;

}

## **5. Видалення / додавання ребра**

void addEdge(vector<vector<int>>& adj, int u, int v) {

adj[u].push\_back(v);

adj[v].push\_back(u);

}

void removeEdge(vector<vector<int>>& adj, int u, int v) {

adj[u].erase(remove(adj[u].begin(), adj[u].end(), v), adj[u].end());

adj[v].erase(remove(adj[v].begin(), adj[v].end(), u), adj[v].end());

}

## **7. Кількість вершин певного степеня**

int countDegreeK(vector<vector<int>>& adj, int k) {

int cnt=0;

for (int i=0;i<adj.size();i++) {

if (adj[i].size()==k) cnt++;

}

return cnt;

}

## **8. BFS для знаходження вершин на відстані d**

vector<int> verticesAtDistance(vector<vector<int>>& adj, int start, int d) {

vector<int> dist(adj.size(), -1);

queue<int> q;

dist[start]=0; q.push(start);

while(!q.empty()) {

int v=q.front(); q.pop();

for (int u: adj[v]) if (dist[u]==-1) {

dist[u]=dist[v]+1;

q.push(u);

}

}

vector<int> res;

for (int i=0;i<adj.size();i++) if (dist[i]==d) res.push\_back(i);

return res;

}

**Розріджені матриці (послідовно-зв’язне індексне зберігання)**

У білетах майже завжди матриця вигляду B[n,m], але більшість елементів = 0.  
 Тому її стискають так: зберігаються тільки ненульові значення.

**Базова структура (рядок, стовпець, значення + вказівник):**

struct Elem {

int row, col, val;

Elem\* next;

};

Elem\* head = NULL;

### **1. Побудова стислого зберігання з 2D масиву**

void buildSparse(int B[20][50], int n, int m, Elem\*& head) {

head = NULL;

Elem\* tail = NULL;

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

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

if (B[i][j] != 0) {

Elem\* node = new Elem{i, j, B[i][j], NULL};

if (!head) head = node;

else tail->next = node;

tail = node;

}

}

}

}

### **2. Копіювання стислої матриці**

Elem\* copySparse(Elem\* head) {

if (!head) return NULL;

Elem\* newHead = NULL, \*tail = NULL;

for (Elem\* cur=head; cur; cur=cur->next) {

Elem\* node = new Elem{cur->row, cur->col, cur->val, NULL};

if (!newHead) newHead = node;

else tail->next = node;

tail = node;

}

return newHead;

}

### **3. Знаходження максимального елемента**

int maxSparse(Elem\* head) {

if (!head) return -1e9;

int mx = head->val;

for (Elem\* cur=head; cur; cur=cur->next)

if (cur->val > mx) mx = cur->val;

return mx;

}

### **4. Номер рядка з найбільшою кількістю нулів**

(тобто найменша кількість ненульових елементів)

int rowWithMostZeros(Elem\* head, int n, int m) {

int count[100]={0};

for (Elem\* cur=head; cur; cur=cur->next) {

count[cur->row]++;

}

int bestRow=0, bestZeros=m-count[0];

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

int zeros=m-count[i];

if (zeros>bestZeros){

bestZeros=zeros;

bestRow=i;

}

}

return bestRow;

}

### **5. Середнє арифметичне ненульових елементів**

double avgSparse(Elem\* head) {

int sum=0, cnt=0;

for (Elem\* cur=head; cur; cur=cur->next) {

sum+=cur->val;

cnt++;

}

if (cnt==0) return 0;

return (double)sum/cnt;

}

### **6. Друк усіх елементів (стисле представлення)**

void printSparse(Elem\* head) {

for (Elem\* cur=head; cur; cur=cur->next) {

cout << "(" << cur->row << "," << cur->col << ")=" << cur->val << " ";

}

cout << endl;

}

### **7. Пошук i-го елемента в стислому зберіганні (за порядком у списку)**

int getIth(Elem\* head, int idx) {

int k=0;

for (Elem\* cur=head; cur; cur=cur->next) {

if (k==idx) return cur->val;

k++;

}

return 0; // якщо індекс більший ніж довжина списку

}

### **8. Знаходження середнього арифметичного рядка**

double avgRow(Elem\* head, int row, int m) {

int sum=0, cnt=0;

for (Elem\* cur=head; cur; cur=cur->next) {

if (cur->row==row) {

sum+=cur->val;

cnt++;

}

}

return cnt==0 ? 0 : (double)sum/m;

}

# **Матриці, поліноми, стекові задачі**

# **Поліноми (списки коефіцієнтів)**

### **1. Похідна полінома**

struct Term {

int coef, power;

Term\* next;

};

Term\* derivative(Term\* head) {

Term\* res=nullptr, \*tail=nullptr;

for (Term\* t=head; t; t=t->next) {

if (t->power>0) {

Term\* node = new Term{t->coef\*t->power, t->power-1, nullptr};

if (!res) res=node;

else tail->next=node;

tail=node;

}

}

return res;

}

### **2. Сума двох поліномів**

Term\* addPoly(Term\* a, Term\* b) {

Term\* res=nullptr,\*tail=nullptr;

while(a||b) {

int c=0,p=0;

if (b==nullptr || (a && a->power<b->power)) { c=a->coef; p=a->power; a=a->next; }

else if (a==nullptr || (b && b->power<a->power)) { c=b->coef; p=b->power; b=b->next; }

else { c=a->coef+b->coef; p=a->power; a=a->next; b=b->next; }

if (c!=0) {

Term\* node=new Term{c,p,nullptr};

if (!res) res=node;

else tail->next=node;

tail=node;

}

}

return res;

}

### **3. Різниця двох поліномів**

(аналогічно сумі, тільки c=a->coef-b->coef)

## **🔹 Стекові задачі**

### **1. Реалізація стека**

struct Stack {

int a[1000],top=-1;

void push(int x){a[++top]=x;}

int pop(){return a[top--];}

bool empty(){return top<0;}

};

### **2. Обчислення ПОЛІЗ (постфіксного виразу)**

int evalPoliz(string expr) {

Stack st;

for (char c: expr) {

if (isdigit(c)) st.push(c-'0');

else {

int b=st.pop(),a=st.pop();

if (c=='+') st.push(a+b);

if (c=='-') st.push(a-b);

if (c=='\*') st.push(a\*b);

if (c=='/') st.push(a/b);

}

}

return st.pop();

}

### **3. Ітеративна функція Аккермана / F(m,n) через стек**

(схема однакова — замінюємо рекурсію стеком):

int F(int m,int n){

struct State{int m,n;};

Stack st;

st.push(m); st.push(n);

int res=0;

// далі імітація рекурсії через while(!st.empty())

}

**4. Масив зі 100 елементів використовується для реалізації черги. На вході задана послідовність невідʼємних цілих чисел. Якщо число непарне — додається в чергу, якщо парне — елемент вилучається з черги. Потрібно обробляти "аварійні" ситуації (переповнення або вилучення з порожньої черги).**

struct Queue {

int data[MAX\_SIZE];

int front, rear, count;

};

void init(Queue &q) {

q.front = 0;

q.rear = -1;

q.count = 0;

}

bool isEmpty(Queue &q) {

return q.count == 0;

}

bool isFull(Queue &q) {

return q.count == MAX\_SIZE;

}

void enqueue(Queue &q, int x) {

if (isFull(q)) {

cout << "Аварія: черга переповнена!\n";

return;

}

q.rear = (q.rear + 1) % MAX\_SIZE;

q.data[q.rear] = x;

q.count++;

}

void dequeue(Queue &q) {

if (isEmpty(q)) {

cout << "Аварія: черга порожня!\n";

return;

}

q.front = (q.front + 1) % MAX\_SIZE;

q.count--;

}

void printQueue(Queue &q) {

if (isEmpty(q)) {

cout << "Черга порожня.\n";

return;

}

cout << "Черга: ";

for (int i = 0; i < q.count; i++) {

int index = (q.front + i) % MAX\_SIZE;

cout << q.data[index] << " ";

}

cout << endl;

}