Міністерство освіти і науки України

Національний університет «Львівська політехніка»

Кафедра систем штучного інтелекту

A blue and white logo

Description automatically generated

**Звіт**

**про виконання лабораторних та практичних робіт блоку № 6**

На тему: «Динамічні структури (Черга, Стек, Списки, Дерево). Алгоритми обробки динамічних структур.»

***з дисципліни:*** «Основи програмування»

до:

ВНС Лабораторної Роботи № 10

Алготестер Лабораторної Роботи № 5

Львів 2024

### **Тема роботи:**

Динамічні структури (Черга, Стек, Списки, Дерево). Алгоритми обробки динамічних структур

### **Мета роботи:**

Ознайомитись з основними струкрурами данних , навчитись ітеруватись по ним , ознайомитись з алгоритмами їх обробки.

### **Джерела:**

книга - Stephen Prata - “ *C++ Primer Plus ”*

книга *- Aditya Y.Bhargava - “ Grokking algorithms ”*

### **Виконання роботи:**

### **Завдання № 3**

**Requirements :**

Vns Lab 10

**Time:**

**Expected: 1 hour**

**Spent: up to 1 hour**

#include <iostream>

#include <fstream>

#include <vector>

#include <string>

#include <limits>

#include <sstream>

template <class T>

class LinkedList {

private:

struct Node {

T value;

Node\* next;

Node\* prev;

Node(T value, Node\* next = nullptr, Node\* prev = nullptr)

: value(value), next(next), prev(prev) {}

};

Node\* head = nullptr;

Node\* tail = nullptr;

std::size\_t size;

public:

LinkedList() : head(nullptr), tail(nullptr), size(0) {}

~LinkedList() {

clear();

}

void push\_back(T value) {

Node\* newNode = new Node(value);

if (!head) {

head = tail = newNode;

}

else {

tail->next = newNode;

newNode->prev = tail;

tail = newNode;

}

size++;

}

void print() const {

if (!head) {

std::cout << "Empty List!" << std::endl;

return;

}

Node\* temp = head;

while (temp) {

std::cout << temp->value << " ";

temp = temp->next;

}

std::cout << "\b";

std::cout << std::endl;

}

void save\_to\_file(const std::string& filename) const {

std::ofstream file(filename);

if (!file) {

throw std::ios\_base::failure("Failed to create or open the file: " + filename);

}

Node\* temp = head;

while (temp) {

file << temp->value << " ";

temp = temp->next;

}

file.close();

}

void load\_from\_file(const std::string& filename) {

clear();

std::ifstream file(filename);

if (!file) { // Check if the file was opened successfully

throw std::ios\_base::failure("Failed to create or open the file: " + filename);

}

T value;

while (file >> value) {

push\_back(value);

}

file.close();

}

void clear() {

Node\* current = head;

while (current) {

Node\* temp = current;

current = current->next;

delete temp;

}

head = tail = nullptr;

size = 0;

}

void insert(int index, T el) {

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

throw std::out\_of\_range("Index out of bounds");

}

Node\* newNode = new Node(el);

if (index == 0) {

newNode->next = head;

if (head) {

head->prev = newNode;

}

head = newNode;

if (size == 0) {

tail = newNode;

}

}

else if (index == size) {

newNode->prev = tail;

tail->next = newNode;

tail = newNode;

}

else {

Node\* current = head;

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

current = current->next;

}

newNode->next = current->next;

newNode->prev = current;

current->next->prev = newNode;

current->next = newNode;

}

size++;

}

void insert(int index, int elCount, T\* els) {

if (elCount <= 0) {

return;

}

Node\* newHead = new Node(els[0]);

Node\* current = newHead;

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

current->next = new Node(els[i], nullptr, current);

current = current->next;

}

if (index == 0) {

if (head) {

current->next = head;

head->prev = current;

}

head = newHead;

if (size == 0) {

tail = current;

}

}

else {

Node\* prev = head;

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

prev = prev->next;

}

current->next = prev->next;

if (prev->next) {

prev->next->prev = current;

}

prev->next = newHead;

newHead->prev = prev;

if (index == size) {

tail = current;

}

}

size += elCount;

}

void erase(int index, int count) {

if (index == 0) {

Node\* temp = head;

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

Node\* next = temp->next;

delete temp;

temp = next;

}

head = temp;

if (head) {

head->prev = nullptr;

}

else {

tail = nullptr;

}

}

else {

Node\* prev = head;

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

prev = prev->next;

}

Node\* current = prev->next;

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

Node\* next = current->next;

delete current;

current = next;

}

prev->next = current;

if (current) {

current->prev = prev;

}

else {

tail = prev;

}

}

size -= count;

}

std::size\_t get\_size()

{

return this->size;

}

};

int main() {

LinkedList<int> list;

std::cout << "Test Case 1: Inserting elements 1, 2, 3, 4, 5" << std::endl;

for (int i = 1; i <= 5; ++i) {

list.push\_back(i);

}

list.print();

std::cout << "Test Case 2: Inserting element 10 at index 2" << std::endl;

list.insert(2, 10);

list.print();

std::cout << "Test Case 3: Erasing 2 elements starting from index 2" << std::endl;

list.erase(2, 2);

list.print();

std::cout << "Test Case 4: Checking size of the list" << std::endl;

std::cout << "Size: " << list.get\_size() << std::endl;

std::cout << "Test Case 5: Saving list to file 'list.txt'" << std::endl;

list.save\_to\_file("D:\\nulp\\OP\\AllEpics\\Sixth\\list.txt"); // Explicit path ( Vs code g++ specific )

std::cout << "Test Case 6: Clearing the list" << std::endl;

list.clear();

list.print();

std::cout << "Test Case 7: Loading list from file 'list.txt'" << std::endl;

list.load\_from\_file("D:\\nulp\\OP\\AllEpics\\Sixth\\list.txt"); // Explicit path ( Vs code g++ specific )

list.print();

std::cout << "Test Case 8: Erasing element at index 1" << std::endl;

list.erase(1, 1);

list.print();

std::cout << "Test Case 9: Inserting element 6 at the end" << std::endl;

list.push\_back(6);

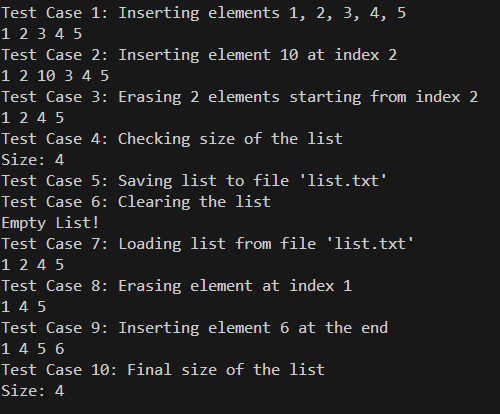
list.print();

std::cout << "Test Case 10: Final size of the list" << std::endl;

std::cout << "Size: " << list.get\_size() << std::endl;

return 0;

}



### **Завдання № 4**

**Requirements :**

Algotester Lab 5

**Time:**

**Expected: 1 hour**

**Spent: up to 3 hours**

#include <iostream>

#include <vector>

#include <queue>

#include <algorithm>

using namespace std;

const int dx[] = { 1, -1, 0, 0 };

const int dy[] = { 0, 0, 1, -1 };

int main() {

int N, M;

cin >> N >> M;

int x, y;

cin >> x >> y;

x--; y--;

vector<vector<int>> height(N, vector<int>(M, -1));

queue<pair<int, int>> q;

q.push({ x, y });

height[x][y] = 0;

while (!q.empty()) {

auto fEl = q.front();

q.pop();

for (int i = 0; i < 4; ++i) {

int nx = fEl.first + dx[i];

int ny = fEl.second + dy[i];

if (nx >= 0 && nx < N && ny >= 0 && ny < M && height[nx][ny] == -1) {

height[nx][ny] = height[fEl.first][fEl.second] + 1;

q.push({ nx, ny });

}

}

}

int maxHeight = 0;

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

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

maxHeight = max(maxHeight, height[i][j]);

}

}

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

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

cout << maxHeight - height[i][j] << " ";

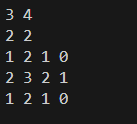
}

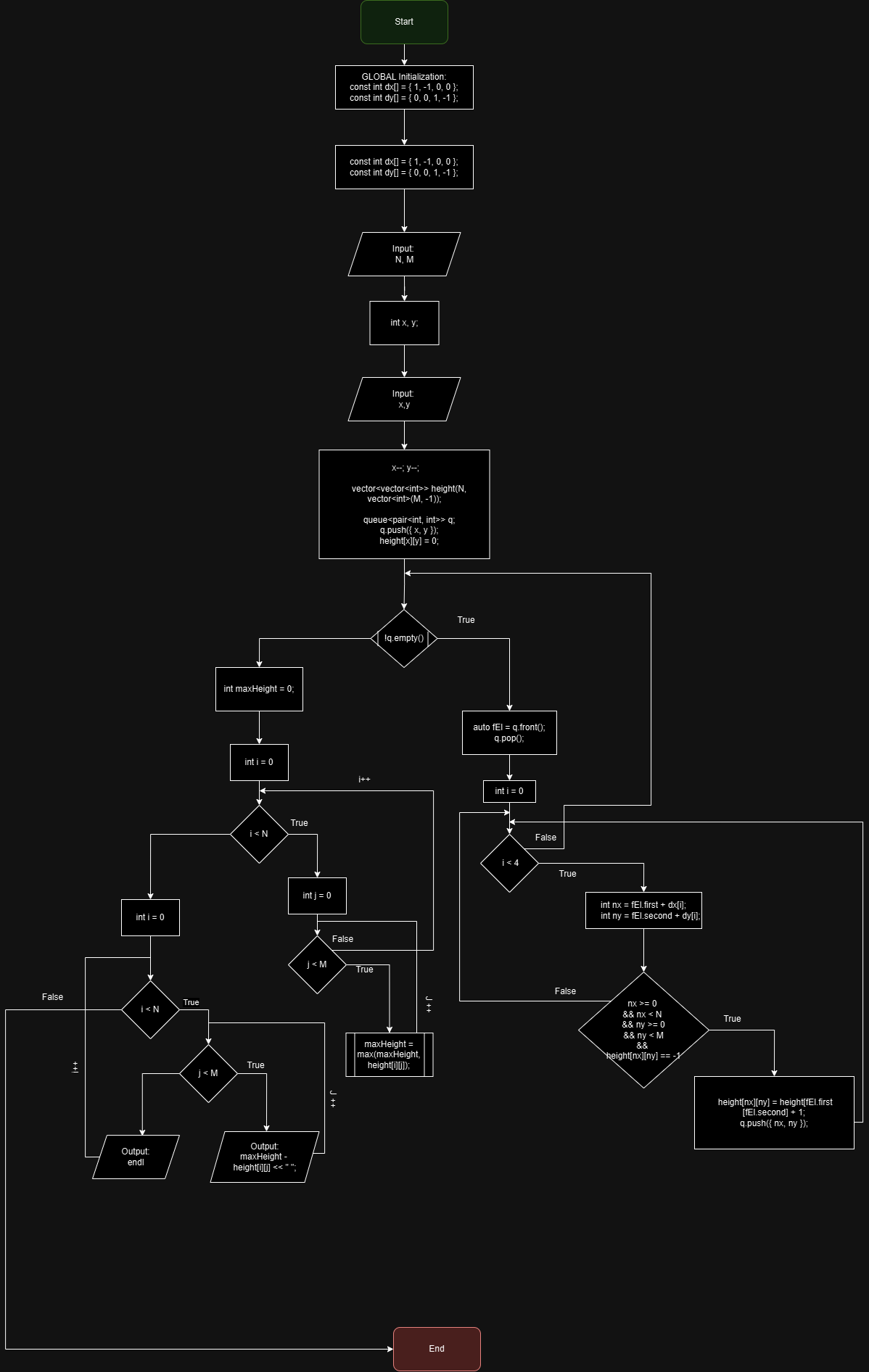
cout << endl;

}

return 0;

}





### **Завдання № 5**

**Requirements :**

Algotester Lab 78 ( v1 , v3 )

**Time:**

**Expected: 1 hour**

**Spent: up to 3 hours**

**V1:**

#include <iostream>

#include <vector>

#include <string>

#include <sstream>

enum class operationE {

insert, erase, size, get, set, print, none

};

operationE parse(const std::string& str) {

if (str == "erase") return operationE::erase;

if (str == "insert") return operationE::insert;

if (str == "size") return operationE::size;

if (str == "get") return operationE::get;

if (str == "set") return operationE::set;

if (str == "print") return operationE::print;

return operationE::none;

}

template <class T>

class LinkedList {

private:

struct Node {

T value;

Node\* next;

Node\* prev;

Node(T value, Node\* next = nullptr, Node\* prev = nullptr)

: value(value), next(next), prev(prev) {}

};

Node\* head;

Node\* tail;

std::size\_t size;

public:

LinkedList() : head(nullptr), tail(nullptr), size(0) {}

~LinkedList() {

clear();

}

void push\_back(T value) {

Node\* newNode = new Node(value);

if (!head) {

head = tail = newNode;

} else {

tail->next = newNode;

newNode->prev = tail;

tail = newNode;

}

++size;

}

void clear() {

Node\* current = head;

while (current) {

Node\* temp = current;

current = current->next;

delete temp;

}

head = tail = nullptr;

size = 0;

}

void insert(int index, int elCount, const std::vector<T>& els) {

if (index < 0 || index > size || elCount <= 0) return;

Node\* newHead = new Node(els[0]);

Node\* current = newHead;

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

current->next = new Node(els[i], nullptr, current);

current = current->next;

}

if (index == 0) {

if (head) {

current->next = head;

head->prev = current;

}

head = newHead;

if (size == 0) {

tail = current;

}

} else {

Node\* prev = head;

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

prev = prev->next;

}

current->next = prev->next;

if (prev->next) {

prev->next->prev = current;

}

prev->next = newHead;

newHead->prev = prev;

if (index == size) {

tail = current;

}

}

size += elCount;

}

void erase(int index, int count) {

if (index < 0 || index >= size || count <= 0) return;

if (index == 0) {

Node\* temp = head;

for (int i = 0; i < count && temp; ++i) {

Node\* next = temp->next;

delete temp;

temp = next;

--size;

}

head = temp;

if (head) {

head->prev = nullptr;

} else {

tail = nullptr;

}

} else {

Node\* prev = head;

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

prev = prev->next;

}

Node\* current = prev->next;

for (int i = 0; i < count && current; ++i) {

Node\* next = current->next;

delete current;

current = next;

--size;

}

prev->next = current;

if (current) {

current->prev = prev;

} else {

tail = prev;

}

}

}

std::size\_t get\_size() const {

return size;

}

T get(std::size\_t index) const {

if (index >= size) throw std::out\_of\_range("Index out of bounds");

Node\* tmpStep = head;

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

tmpStep = tmpStep->next;

}

return tmpStep->value;

}

void set(std::size\_t index, T val) {

if (index >= size) throw std::out\_of\_range("Index out of bounds");

Node\* tmpStep = head;

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

tmpStep = tmpStep->next;

}

tmpStep->value = val;

}

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

Node\* temp = list.head;

while (temp != nullptr) {

out << temp->value << ' ';

temp = temp->next;

}

return out;

}

};

int main() {

LinkedList<int> list;

unsigned int Q;

std::cin >> Q;

std::string tmpOpStr;

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

std::cin >> tmpOpStr;

operationE parseOp = parse(tmpOpStr);

switch (parseOp) {

case operationE::insert: {

int index, n;

std::cin >> index >> n;

std::vector<int> N(n);

for (int& num : N) {

std::cin >> num;

}

list.insert(index, n, N);

break;

}

case operationE::size:

std::cout << list.get\_size() << std::endl;

break;

case operationE::get: {

int index;

std::cin >> index;

std::cout << list.get(index) << std::endl;

break;

}

case operationE::set: {

int index, value;

std::cin >> index >> value;

list.set(index, value);

break;

}

case operationE::print:

std::cout << list << std::endl;

break;

case operationE::erase: {

int index, count;

std::cin >> index >> count;

list.erase(index, count);

break;

}

default:

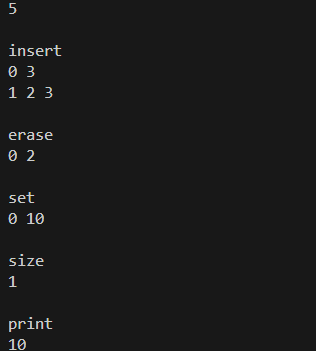
break;

}

}

return 0;

}



**V3:**

#include <iostream>

#include <string>

#include <algorithm>

using namespace std;

enum Operation {

INSERT,

SIZE,

PRINT,

CONTAINS,

UNKNOWN

};

Operation getOperation(const string& command) {

if (command == "insert") return INSERT;

if (command == "size") return SIZE;

if (command == "print") return PRINT;

if (command == "contains") return CONTAINS;

return UNKNOWN;

}

template<typename T>

class Tree {

private:

struct Node {

T value;

Node\* left;

Node\* right;

Node(T val) : value(val), left(nullptr), right(nullptr) {}

};

Node\* root;

int size;

void clear(Node\* root) {

if (root != nullptr) {

clear(root->left);

clear(root->right);

delete root;

}

}

void insertP(Node\* node, T value) {

if (value < node->value) {

if (node->left == nullptr) {

node->left = new Node(value);

size++;

}

else {

insertP(node->left, value);

}

}

else if (value > node->value) {

if (node->right == nullptr) {

node->right = new Node(value);

size++;

}

else {

insertP(node->right, value);

}

}

}

bool containsP(Node\* node, T value) {

if (node == nullptr) return false;

if (value == node->value) return true;

if (value < node->value) return containsP(node->left, value);

return containsP(node->right, value);

}

void printP(Node\* node, ostream& os) const {

if (node != nullptr) {

printP(node->left, os);

os << node->value << " ";

printP(node->right, os);

}

}

public:

Tree() : root(nullptr), size(0) {}

~Tree() {

clear(root);

}

void insert(T value) {

if (root == nullptr) {

root = new Node(value);

size++;

}

else {

insertP(root, value);

}

}

bool contains(T value) {

return containsP(root, value);

}

int getSize() const {

return size;

}

friend ostream& operator<<(ostream& os, const Tree& tree) {

tree.printP(tree.root, os);

return os;

}

};

int main() {

int Q;

cin >> Q;

Tree<int> tree;

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

string option;

cin >> option;

Operation operation = getOperation(option);

switch (operation) {

case INSERT: {

int value;

cin >> value;

tree.insert(value);

break;

}

case SIZE: {

cout << tree.getSize() << endl;

break;

}

case CONTAINS: {

int value;

cin >> value;

cout << (tree.contains(value) ? "Yes" : "No") << endl;

break;

}

case PRINT: {

cout << tree;

break;

}

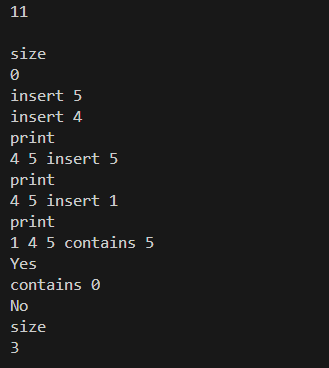
default:

break;

}

}

}



### **Завдання № 6**

**Requirements :**

Class Practice Task

**Time:**

**Expected: 1 hour**

**Spent: up to 1 hour**

#include <iostream>

// Node structure for linked list

struct Node {

int data;

Node\* next;

Node(int val) : data(val), next(nullptr) {}

};

// Function to reverse the linked list

Node\* reverse(Node\* head) {

Node\* prev = nullptr;

Node\* current = head;

Node\* next = nullptr;

while (current != nullptr) {

next = current->next; // Store next node

current->next = prev; // Reverse the link

prev = current; // Move prev and current one step forward

current = next;

}

return prev; // New head of the reversed list

}

// Function to print the linked list

void printList(Node\* head) {

while (head != nullptr) {

std::cout << head->data << " ";

head = head->next;

}

std::cout << std::endl;

}

// Function to compare two linked lists

bool compare(Node\* h1, Node\* h2) {

while (h1 != nullptr && h2 != nullptr) {

if (h1->data != h2->data) {

return false; // Data mismatch

}

h1 = h1->next;

h2 = h2->next;

}

return h1 == nullptr && h2 == nullptr; // Both must reach the end

}

// Function to add two large numbers represented as linked lists

Node\* add(Node\* n1, Node\* n2) {

Node dummy(0); // Dummy node to simplify the process

Node\* current = &dummy;

int carry = 0;

while (n1 != nullptr || n2 != nullptr || carry) {

int sum = carry;

if (n1 != nullptr) {

sum += n1->data;

n1 = n1->next;

}

if (n2 != nullptr) {

sum += n2->data;

n2 = n2->next;

}

carry = sum / 10; // Update carry

current->next = new Node(sum % 10); // Create new node

current = current->next;

}

return dummy.next; // Return the next node to dummy

}

// TreeNode structure for binary tree

struct TreeNode {

int value;

TreeNode\* left;

TreeNode\* right;

TreeNode(int val) : value(val), left(nullptr), right(nullptr) {}

};

// Function to create a mirrored (flipped) copy of the tree

TreeNode\* create\_mirror\_flip(TreeNode\* root) {

if (root == nullptr) return nullptr;

TreeNode\* new\_root = new TreeNode(root->value);

new\_root->left = create\_mirror\_flip(root->right);

new\_root->right = create\_mirror\_flip(root->left);

return new\_root;

}

// Function to print tree nodes in in-order traversal

void printTree(TreeNode\* root) {

if (root != nullptr) {

printTree(root->left);

std::cout << root->value << " ";

printTree(root->right);

}

}

// Function to update tree nodes' values as the sum of their subtrees

void tree\_sum(TreeNode\* root) {

if (root == nullptr) return;

int left\_sum = 0;

int right\_sum = 0;

if (root->left != nullptr) {

left\_sum += root->left->value;

}

if (root->right != nullptr) {

right\_sum += root->right->value;

}

tree\_sum(root->left);

tree\_sum(root->right);

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

root->value = left\_sum + right\_sum;

}

int main() {

// Linked List Operations

Node\* head = new Node(1);

head->next = new Node(2);

head->next->next = new Node(3);

head->next->next->next = new Node(4);

std::cout << "Original Linked List: ";

printList(head);

head = reverse(head);

std::cout << "Reversed Linked List: ";

printList(head);

Node\* list1 = new Node(1);

list1->next = new Node(2);

list1->next->next = new Node(3);

Node\* list2 = new Node(1);

list2->next = new Node(2);

list2->next->next = new Node(3);

std::cout << "Are lists equal? " << (compare(list1, list2) ? "Yes" : "No") << std::endl;

list2->next->next->data = 4;

std::cout << "Are lists equal after modification? " << (compare(list1, list2) ? "Yes" : "No") << std::endl;

Node\* num1 = new Node(9);

num1->next = new Node(9);

num1->next->next = new Node(9);

Node\* num2 = new Node(1);

num2->next = new Node(0);

num2->next->next = new Node(0);

Node\* sum = add(num1, num2);

std::cout << "Sum of numbers: ";

printList(sum);

// Binary Tree Operations

//structure:

// 1

// / \

// 2 3

// / \

// 4 5

TreeNode\* root = new TreeNode(1);

root->left = new TreeNode(2);

root->right = new TreeNode(3);

root->left->left = new TreeNode(4);

root->left->right = new TreeNode(5);

std::cout << "Original Tree: ";

printTree(root);

std::cout << std::endl;

TreeNode\* mirroredRoot = create\_mirror\_flip(root);

std::cout << "Mirrored Tree: ";

printTree(mirroredRoot);

std::cout << std::endl;

// structure:

// 1

// / \

// 2 3

// / \

// 4 5

TreeNode\* sumTreeRoot = new TreeNode(1);

sumTreeRoot->left = new TreeNode(2);

sumTreeRoot->right = new TreeNode(3);

sumTreeRoot->left->left = new TreeNode(4);

sumTreeRoot->left->right = new TreeNode(5);

std::cout << "Tree before summing: ";

printTree(sumTreeRoot);

std::cout << std::endl;

tree\_sum(sumTreeRoot);

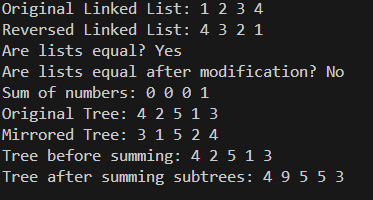
std::cout << "Tree after summing subtrees: ";

printTree(sumTreeRoot);

std::cout << std::endl;

return 0;

}



### **Завдання № 7**

**Requirements :**

Self Practise Task

**Time:**

**Expected: 1 hour**

**Spent: up to 30 mins**

#include <iostream>

#include <algorithm>

int main() {

int n, k;

std::cin >> n >> k;

int\* teeths = new int[n];

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

std::cin >> teeths[i];

}

int maxConsecutive = 0;

int currentConsecutive = 0;

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

if (teeths[i] >= k) {

currentConsecutive++;

maxConsecutive = std::max(maxConsecutive, currentConsecutive);

} else {

currentConsecutive = 0;

}

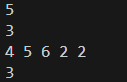
}

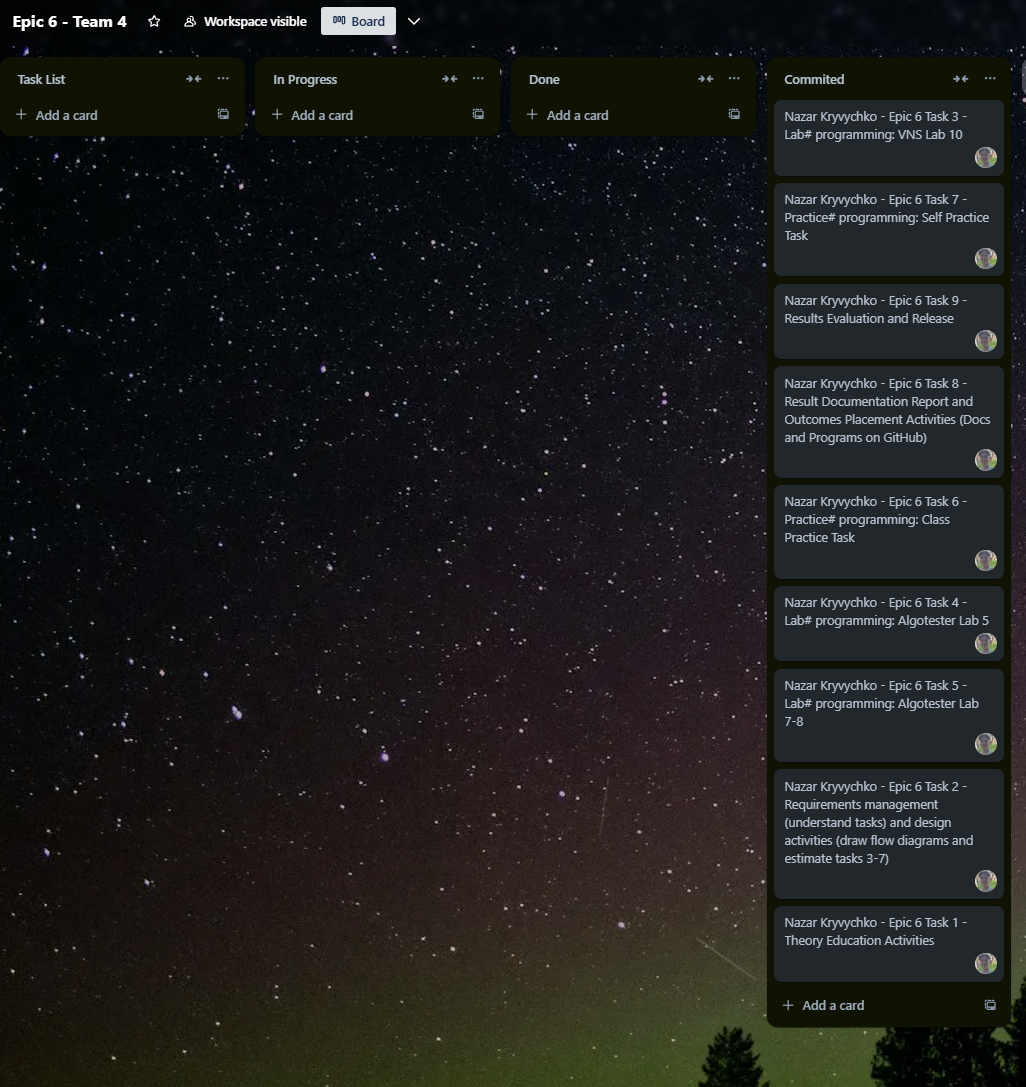
std::cout << maxConsecutive;

delete[] teeths;

return 0;

}



**Trello Configuration & Team meetings:**  


Meeting :

Pull Request: Link

Висновок:  
Я закріпив знання стурктур данних та навчився декілька варіантів ітерації по них.