Міністерство освіти і науки України Національний університет «Львівська політехніка»

Кафедра систем штучного інтелекту



Звіт

про виконання

**Лабораторних та практичних робіт № 6**

***з дисципліни:*** «Мови та парадигми програмування»

***з розділу***: «Динамічні структури (Черга, Стек, Списки, Дерево). Алгоритми обробки динамічних структур»

***Виконав:***

студент групи ШІ-11 Боднар Роман Миколайович

Львів 2023

# Тема роботи:

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

# Мета роботи:

Опанувати роботу з динамічними структурами. Навчитись робити алгоритми обробки динамічних структур. Вивчити теорію та застосування.

# Теоретичні відомості:

Джерела інформації:

<https://studfile.net/preview/3908535/>  
<https://allref.com.ua/uk/skachaty/Dinamichni_strukturi_danih_%28S++%29>

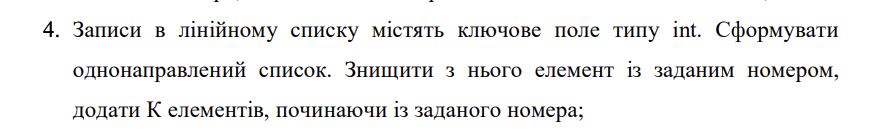
<https://purecodecpp.com/uk/archives/1538>

Статус: Ознайомлений.

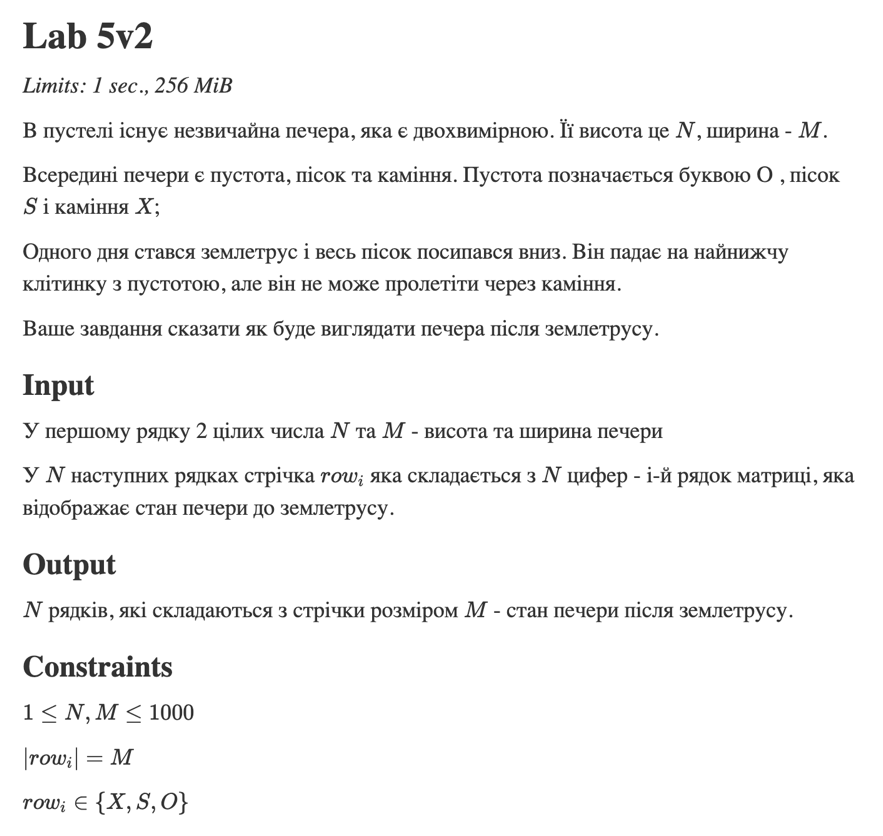
# Виконання роботи:

## Опрацювання завдання та вимог до програм та середовища:

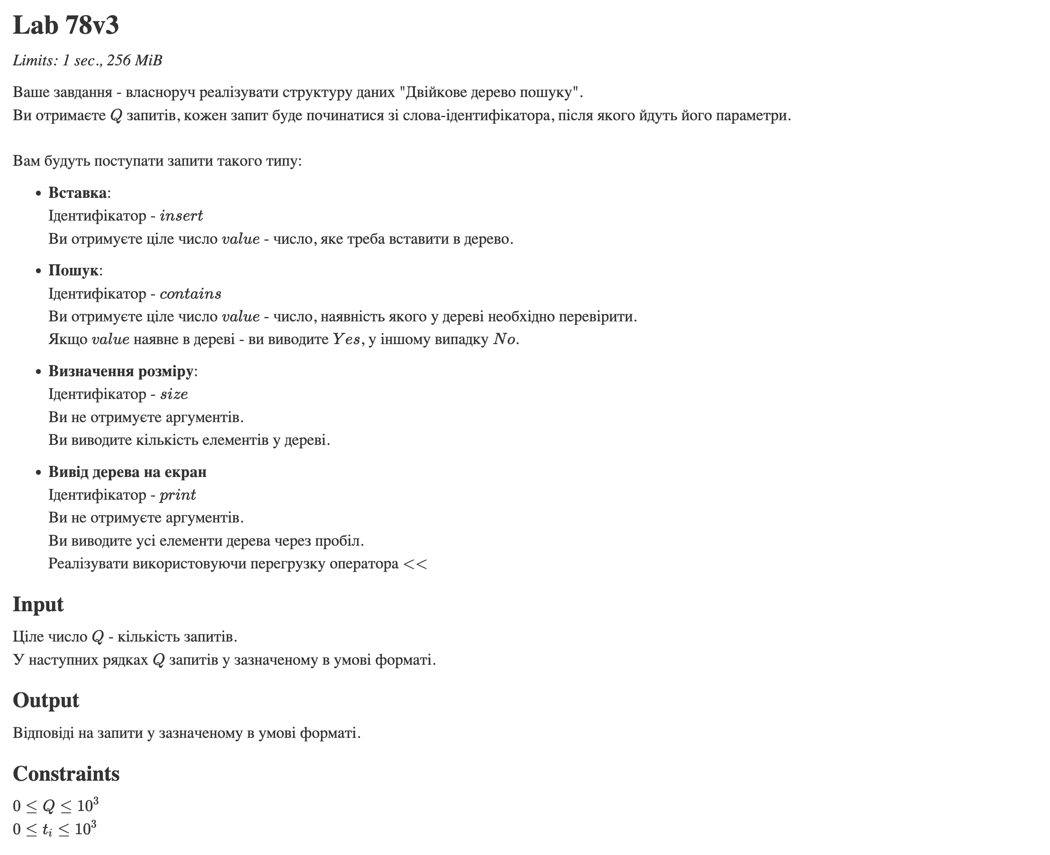
VNS Lab 10 – Variant 4:



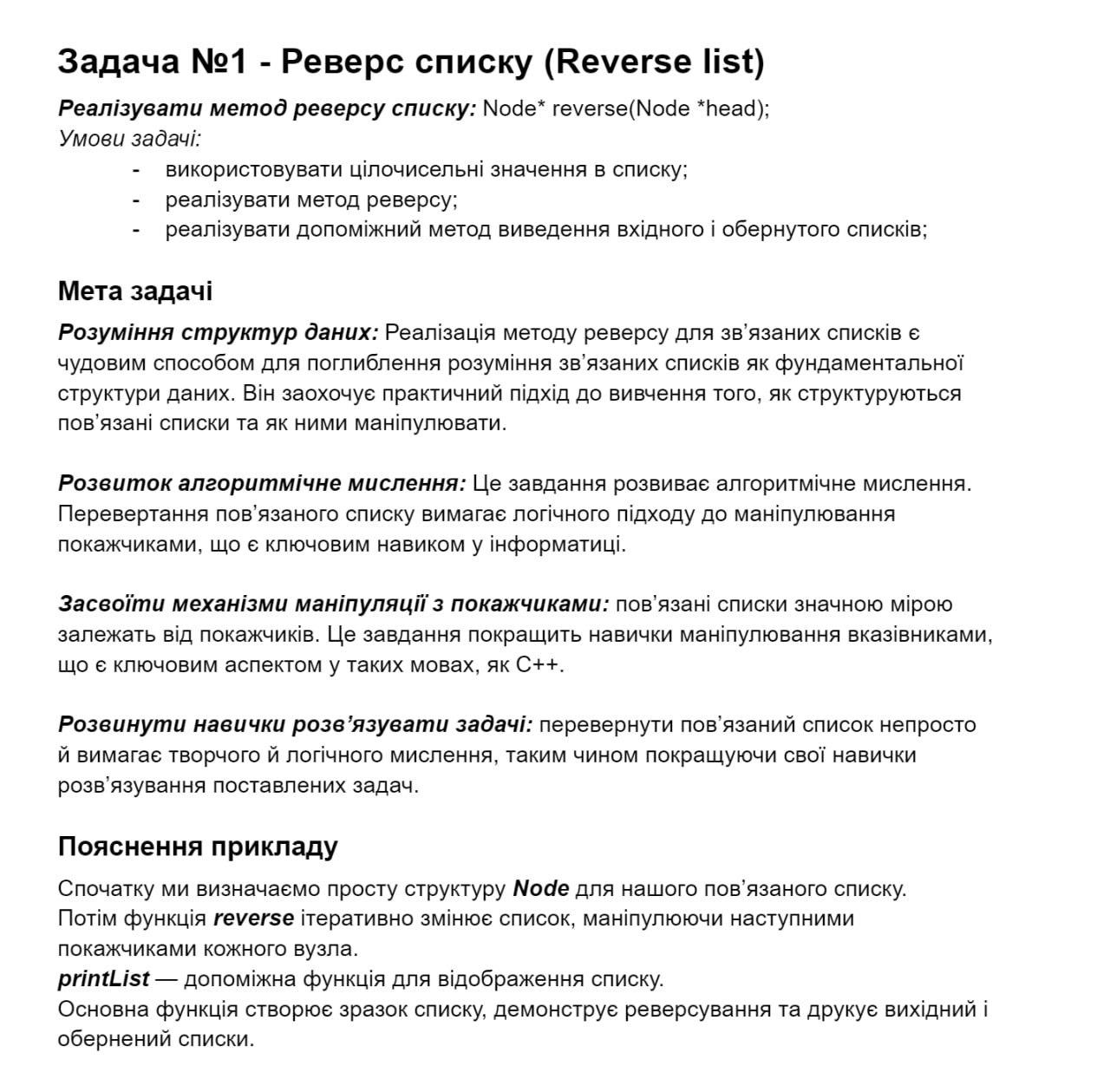
Algotester Lab 5v2:

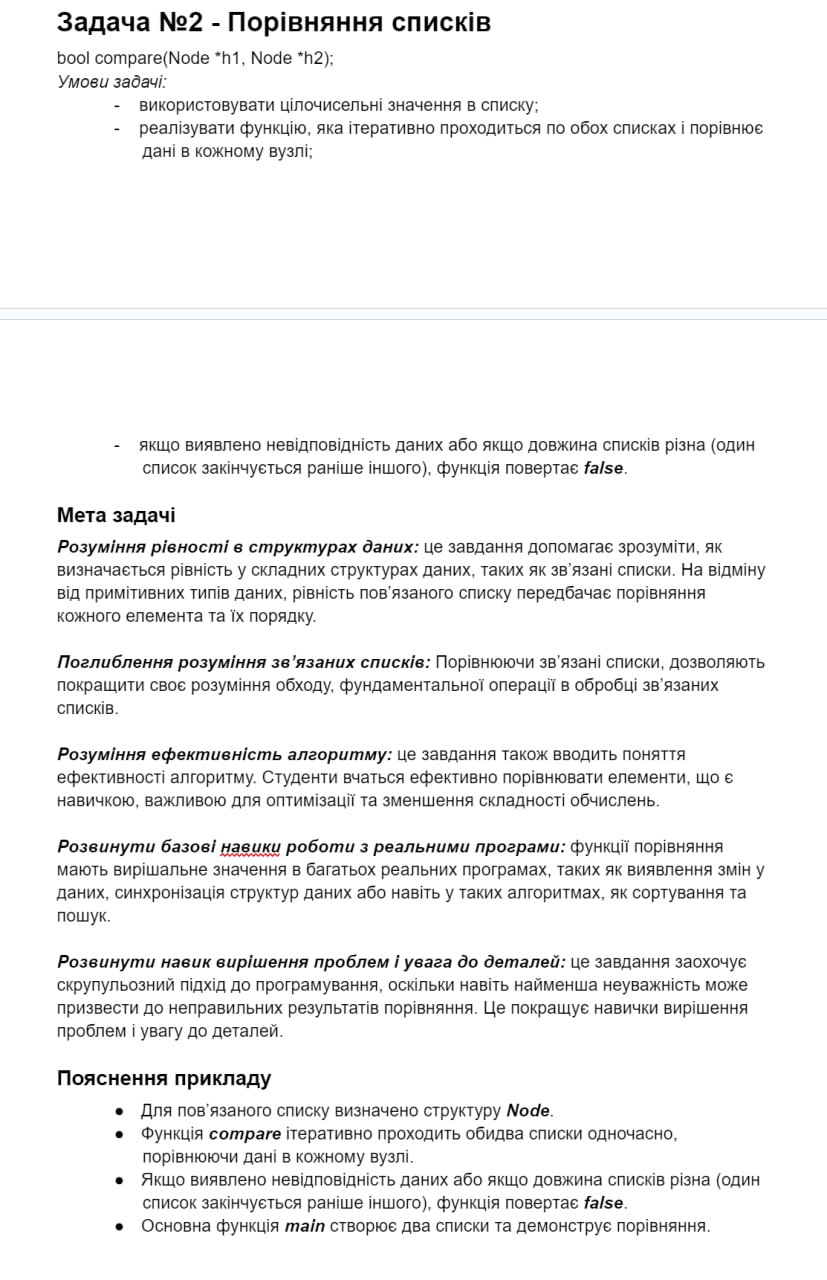


Algotester 78v3:

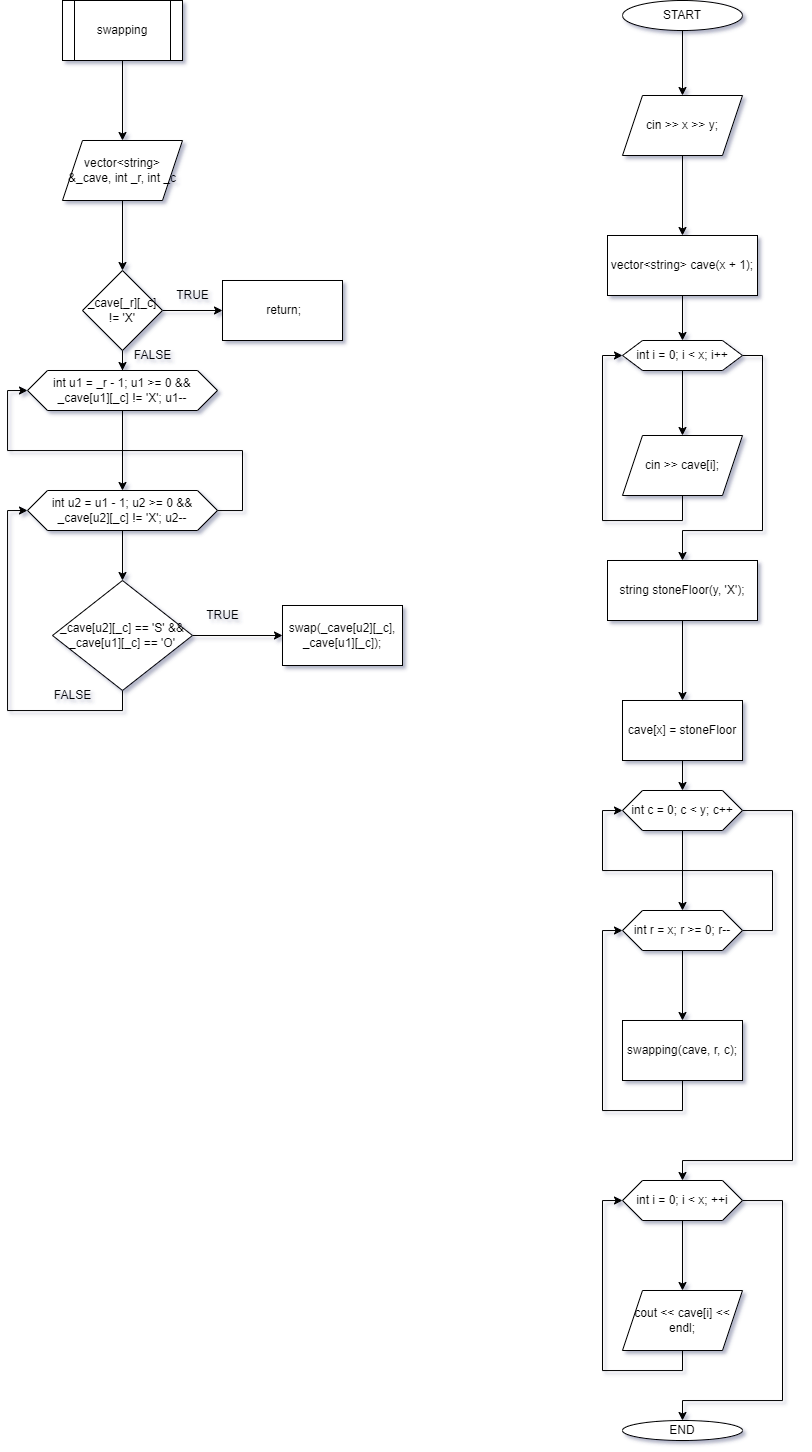


Class practice work:





## Дизайн та планована оцінка часу виконання завдань:



**Algotester Lab 5v2**

## Код програм з посиланням на зовнішні ресурси:

VNS Lab 10 – Variant 4:

#include <iostream>  
#include <fstream>  
using namespace std;  
  
struct point  
{  
 int key;  
 point \*next;  
  
 point(int key, point \*next) : key(key), next(next) {}  
 point() : next(NULL) {}  
 point(int key) : point(key, NULL) {}  
};  
  
void add\_element(point \*\*head, int key, int position)  
{  
 point \*new\_element = new point;  
 new\_element->key = key;  
 new\_element->next = nullptr;  
  
 if (\*head == nullptr)  
 {  
 \*head = new\_element;  
 }  
 else  
 {  
 point \*current = \*head;  
 int i = 0;  
 while (current->next != nullptr && i < position)  
 {  
 current = current->next;  
 i++;  
 }  
 new\_element->next = current->next;  
 current->next = new\_element;  
 }  
}  
  
void remove\_element(point \*\*head, int position)  
{  
 if (\*head == nullptr)  
 {  
 return;  
 }  
 else  
 {  
 point \*current = \*head;  
 int i = 0;  
 while (current->next != nullptr && i < position)  
 {  
 current = current->next;  
 i++;  
 }  
 point \*temp = current->next;  
 current->next = current->next->next;  
 delete temp;  
 }  
}  
  
void print\_list(point \*head)  
{  
 point \*current = head;  
 while (current != nullptr)  
 {  
 cout << current->key << " ";  
 current = current->next;  
 }  
 cout << endl;  
}  
  
void save\_list\_to\_file(point \*head, string filename)  
{  
 ofstream outfile(filename);  
 point \*current = head;  
 while (current != nullptr)  
 {  
 outfile << current->key << endl;  
 current = current->next;  
 }  
 outfile.close();  
}  
  
point \*restore\_list\_from\_file(string filename)  
{  
 ifstream infile(filename);  
 point \*head = nullptr;  
 point \*current = nullptr;  
 int key;  
 infile >> key;  
 while (!infile.eof())  
 {  
 point \*new\_element = new point;  
 new\_element->key = key;  
 new\_element->next = nullptr;  
  
 if (head == nullptr)  
 {  
 head = new\_element;  
 current = head;  
 }  
 else  
 {  
 current->next = new\_element;  
 current = new\_element;  
 }  
  
 infile >> key;  
 }  
 infile.close();  
 return head;  
}  
  
void delete\_list(point \*\*head)  
{  
 point \*current = \*head;  
 while (current != nullptr)  
 {  
 point \*temp = current;  
 current = current->next;  
 delete temp;  
 }  
 \*head = nullptr;  
}  
  
int main()  
{  
 int choice;  
 point \*head = NULL;  
 do  
 {  
 cout << "1. Add element" << endl  
 << "2. Remove element" << endl  
 << "3. Print list" << endl  
 << "4. Save list to file" << endl  
 << "5. Restore list from file" << endl  
 << "0. Exit" << endl  
 << "Enter your choice: ";  
 cin >> choice;  
  
 switch (choice)  
 {  
 case 1:  
 {  
 int key, position;  
 cout << "Enter key: ";  
 cin >> key;  
 cout << "Enter position: ";  
 cin >> position;  
 add\_element(&head, key, position);  
 break;  
 }  
 case 2:  
 {  
 int position;  
 cout << "Enter position: ";  
 cin >> position;  
 remove\_element(&head, position);  
 break;  
 }  
 case 3:  
 print\_list(head);  
 break;  
 case 4:  
 {  
 string filename;  
 cout << "Enter filename: ";  
 cin >> filename;  
 save\_list\_to\_file(head, filename);  
 break;  
 }  
 case 5:  
 {  
 string filename;  
 cout << "Enter filename: ";  
 cin >> filename;  
 head = restore\_list\_from\_file(filename);  
 break;  
 }  
 case 0:  
 delete\_list(&head);  
 break;  
 default:  
 cout << "Wrong choice" << endl;  
 break;  
 }  
  
 } while (choice != 0);  
 return 0;  
}

Algotester Lab 5v2:

#include <iostream>  
#include <vector>  
using namespace std;  
  
void swapping(vector<string> &\_cave, int \_r, int \_c)  
{  
 if (\_cave[\_r][\_c] != 'X') {  
 return;  
 }  
  
 // u - understone  
 for (int u1 = \_r - 1; u1 >= 0 && \_cave[u1][\_c] != 'X'; u1--)  
 for (int u2 = u1 - 1; u2 >= 0 && \_cave[u2][\_c] != 'X'; u2--)  
 if (\_cave[u2][\_c] == 'S' && \_cave[u1][\_c] == 'O')  
 swap(\_cave[u2][\_c], \_cave[u1][\_c]);  
}  
  
int main()  
{  
 int x, y;  
 cin >> x >> y;  
  
 vector<string> cave(x + 1);  
 for (int i = 0; i < x; i++)  
 cin >> cave[i];  
  
 string stoneFloor(y, 'X');  
 cave[x] = stoneFloor;  
  
 // c - column  
 // r - row  
 for (int c = 0; c < y; c++)  
 for (int r = x; r >= 0; r--)  
 swapping(cave, r, c);  
  
 for (int i = 0; i < x; ++i)  
 cout << cave[i] << endl;  
   
 return 0;  
}

Algotester 78v3:

#include <iostream>  
#include <functional>  
using namespace std;  
  
template <typename T>  
class tree {  
private:  
 class BSTNode {  
 public:  
 T value;  
 struct BSTNode\* parent;  
 struct BSTNode\* left;  
 struct BSTNode\* right;  
 BSTNode(T value, BSTNode\* parent, BSTNode\* left, BSTNode\* right)  
 : value(value), parent(parent), left(left), right(right) {}  
 BSTNode() : left(nullptr), right(nullptr), parent(nullptr) {}  
 explicit BSTNode(T value)  
 : value(value), left(nullptr), right(nullptr), parent(nullptr) {}  
 };  
 void print(BSTNode \*node, std::ostream& out) const {  
 BSTNode \*current = node;  
 if (current == nullptr) {  
 return;  
 }  
 print(current->left, out);  
 out << current->value << ' ';  
 print(current->right, out);  
 }  
 void putNode(BSTNode\* bstNode, T \_val) {  
 if (bstNode == nullptr) return;  
 if (bstNode->value == \_val) return;  
 if (\_val < bstNode->value) {  
 if (bstNode->left == nullptr)  
 bstNode->left = new BSTNode(\_val, bstNode, nullptr, nullptr);  
 else putNode(bstNode->left, \_val);  
 } else {  
 if (bstNode->right == nullptr)  
 bstNode->right = new BSTNode(\_val, bstNode, nullptr, nullptr);  
 else putNode(bstNode->right, \_val);  
 }  
 }  
 void removeTree(BSTNode\* bstNode) {  
 if (bstNode == nullptr) return;  
 removeTree(bstNode->left);  
 removeTree(bstNode->right);  
 delete bstNode;  
 }  
 BSTNode\* findNode(T \_val) {  
 BSTNode\* pBstNode = head;  
 while (pBstNode != nullptr) {  
 if (pBstNode->value == \_val) return pBstNode;  
 if (\_val < pBstNode->value) pBstNode = pBstNode->left;  
 else pBstNode = pBstNode->right;  
 }  
 return nullptr;  
 }  
 void appendNode(BSTNode\* bstNode, BSTNode\* localHead) {  
 if (bstNode == nullptr) return;  
 if (localHead == nullptr) {  
 head = bstNode;  
 sizeValue++;  
 return;  
 }  
 if (bstNode->value < localHead->value) {  
 if (localHead->left == nullptr) {  
 localHead->left = bstNode;  
 bstNode->parent = localHead;  
 sizeValue++;  
 } else appendNode(bstNode, localHead->left);  
 } else if (bstNode->value > localHead->value) {  
 if (localHead->right == nullptr) {  
 localHead->right = bstNode;  
 bstNode->parent = localHead;  
 sizeValue++;  
 } else appendNode(bstNode, localHead->right);  
 }  
 }  
 void removeNode(BSTNode\* bstNode) {  
 if (bstNode->parent == nullptr) {  
 head = nullptr;  
 BSTNode\* left = bstNode->left, \*right = bstNode->right;  
 delete bstNode;  
 if (left != nullptr) {  
 head = left;  
 left->parent = nullptr;  
 appendNode(right, head);  
 } else if (right != nullptr) {  
 head = right;  
 right->parent = nullptr;  
 }  
 sizeValue--;  
 return;  
 }  
 if (bstNode->parent->left == bstNode) bstNode->parent->left = nullptr;  
 else bstNode->parent->right = nullptr;  
 BSTNode\* left = bstNode->left, \*right = bstNode->right;  
 delete bstNode;  
 sizeValue--;  
 appendNode(left, head);  
 appendNode(right, head);  
 }  
  
 BSTNode\* head;  
 int sizeValue;  
  
public:  
 tree() {  
 head = nullptr;  
 sizeValue = 0;  
 }  
 ~tree() {  
 removeTree(head);  
 }  
  
 int size() {  
 return sizeValue;  
 }  
 void put(T value) {  
 appendNode(new BSTNode(value), head);  
 }  
 bool has(T value) {  
 return findNode(value) != nullptr;  
 }  
 void remove(T value) {  
 BSTNode\* node = findNode(value);  
 if (node == nullptr) return;  
 removeNode(node);  
 }  
  
  
 friend ostream& operator<<(ostream& os, tree<T>& bst) {  
 bst.print(bst.head, os);  
 return os;  
 }  
};  
  
int main() {  
 tree<int> tree;  
 int n;  
 string str\_input;  
 cin >> n;  
 int int\_input;  
 for (int i = 0; i < n; i++) {  
 cin >> str\_input;  
 if (str\_input == "insert") {  
 cin >> int\_input;  
 tree.put(int\_input);  
 } else if (str\_input == "size") {  
 cout << tree.size() << endl;  
 } else if (str\_input == "print") {  
 cout << tree << endl;  
 } else if (str\_input == "contains") {  
 cin >> int\_input;  
 cout << (tree.has(int\_input) ? "Yes" : "No") << endl;  
 }  
 }  
 return 0;  
}

Class practice work:

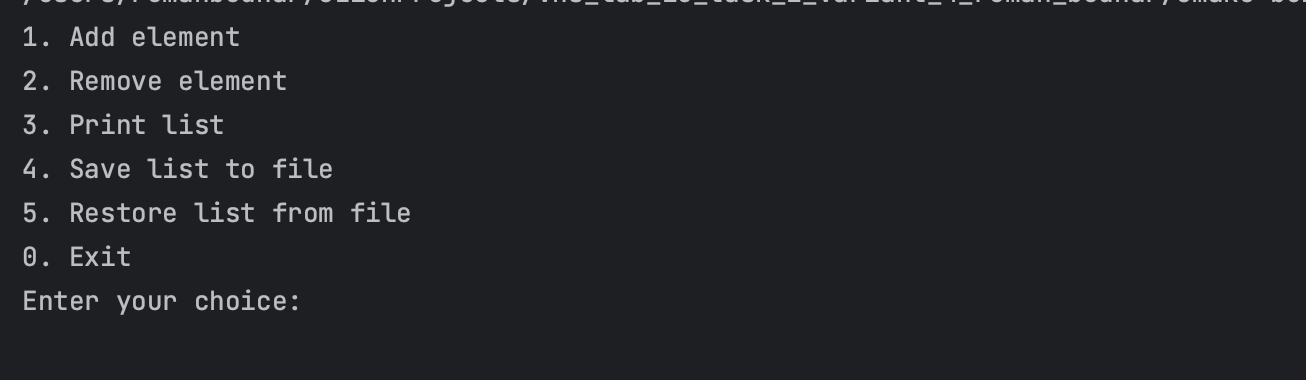
#include <iostream>  
#include <random>  
using namespace std;  
  
class Node  
{  
public:  
 int Value;  
 Node \*Next;  
 Node \*Previous;  
 Node(int value, Node \*next, Node \*prev) : Value(value), Next(next), Previous(prev) {}  
 Node() : Next(NULL), Previous(Previous) {}  
 Node(int value) : Node(value, NULL, NULL) {}  
};  
  
Node \*grabTail(Node \*head)  
{  
 if (head == NULL)  
 return NULL;  
 Node \*pHead = head;  
 while (pHead->Next != NULL)  
 pHead = pHead->Next;  
 return pHead;  
}  
  
Node \*create(int value)  
{  
 Node \*newNode = new Node(value);  
 return newNode;  
}  
  
Node \*push(Node \*head, int value)  
{  
 if (head == NULL)  
 return create(value);  
  
 Node \*pHead = head;  
 Node \*pTail = grabTail(head);  
 Node \*newNode = new Node(value);  
 pTail->Next = newNode;  
 newNode->Previous = pTail;  
 return pHead;  
}  
  
Node \*pop(Node \*head)  
{  
 Node \*pHead = head;  
 Node \*pTail = grabTail(head);  
 pTail->Previous->Next = NULL;  
 delete pTail;  
 return pHead;  
}  
  
Node \*deleteList(Node \*head)  
{  
 Node \*pHead = head;  
 while (pHead != NULL)  
 {  
 Node \*pTemp = pHead;  
 pHead = pHead->Next;  
 delete pTemp;  
 }  
 return NULL;  
}  
  
void printList(Node \*head)  
{  
 Node \*pHead = head;  
 while (pHead != NULL)  
 {  
 cout << pHead->Value << " ";  
 pHead = pHead->Next;  
 }  
 cout << endl;  
}  
  
Node \*reverse(Node \*head)  
{  
 Node \*pCurrent = head, \*pNext = head->Next, \*pTail = grabTail(head);  
 while (pCurrent != NULL)  
 {  
 // swap next and prev pointers  
 swap(pCurrent->Next, pCurrent->Previous);  
 pCurrent = pNext;  
 if (pNext != NULL)  
 pNext = pNext->Next;  
 }  
  
 return pTail;  
}  
  
// compare two lists  
bool compare(Node \*h1, Node \*h2)  
{  
 Node \*pHead1 = h1;  
 Node \*pHead2 = h2;  
 while (pHead1 != NULL && pHead2 != NULL)  
 {  
 if (pHead1->Value != pHead2->Value)  
 return false;  
 pHead1 = pHead1->Next;  
 pHead2 = pHead2->Next;  
 }  
 if (pHead1 != NULL || pHead2 != NULL)  
 return false;  
  
 return true;  
}  
  
Node \*randomList(uniform\_int\_distribution<int> &d, int size = 10)  
{  
 random\_device rd;  
 mt19937 generator(rd());  
 Node \*head = create(d(generator));  
 for (int i = 2; i <= size; i++)  
 head = push(head, d(generator));  
  
 return head;  
}  
  
Node \*add(Node \*n1, Node \*n2)  
{  
 Node \*pTail1 = grabTail(n1), \*pTail2 = grabTail(n2);  
  
 int sum = 0, carry = 0;  
 Node \*sumList = NULL;  
  
 while (pTail1 != NULL || pTail2 != NULL)  
 {  
 sum = (pTail1 != NULL ? pTail1->Value : 0) +  
 (pTail2 != NULL ? pTail2->Value : 0) +  
 carry;  
 sumList = push(sumList, sum % 10);  
 carry = sum / 10;  
 if (pTail1)  
 pTail1 = pTail1->Previous;  
 if (pTail2)  
 pTail2 = pTail2->Previous;  
 }  
  
 if (carry != 0)  
 sumList = push(sumList, carry);  
  
 sumList = reverse(sumList);  
 return sumList;  
}  
  
int main()  
{  
 uniform\_int\_distribution<> dist(1, 100);  
 Node \*head1, \*head2;  
  
 cout << "Task 1\n";  
 head1 = randomList(dist);  
 cout << "Before reverse: ";  
 printList(head1);  
 cout << "After reverse: ";  
 head1 = reverse(head1);  
 printList(head1);  
  
 cout << "\nTask 2\n";  
 head2 = randomList(dist);  
 cout << "List 1: ";  
 printList(head1);  
 cout << "List 2: ";  
 printList(head2);  
 cout << "Compare: ";  
 if (compare(head1, head2))  
 cout << "Lists are equal\n";  
 else  
 cout << "Lists are not equal\n";  
 cout << "List 3: ";  
 printList(head2);  
 cout << "List 4: ";  
 printList(head2);  
 cout << "Compare: ";  
 if (compare(head2, head2))  
 cout << "Lists are equal\n";  
 else  
 cout << "Lists are not equal\n";  
 head1 = deleteList(head1);  
 head2 = deleteList(head2);  
  
 cout << "\nTask 3\n";  
 uniform\_int\_distribution<> dist2(0, 9);  
 head1 = randomList(dist2, 8);  
 head2 = randomList(dist2);  
 cout << "List 1: ";  
 printList(head1);  
 cout << "List 2: ";  
 printList(head2);  
  
 cout << "Sum: ";  
 Node \*sum = add(head1, head2);  
 printList(sum);  
  
 head1 = deleteList(head1);  
 head2 = deleteList(head2);  
 sum = deleteList(sum);  
 return 0;  
}

1.2:

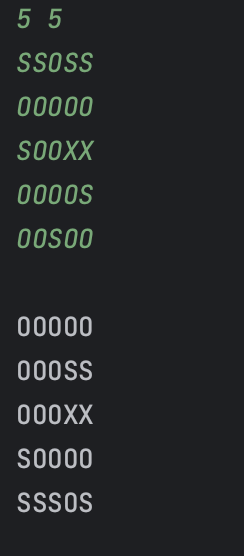
#include <iostream>  
#include <random>  
using namespace std;  
  
struct TreeNode  
{  
 int Value;  
 TreeNode \*Left;  
 TreeNode \*Right;  
 TreeNode \*Parent;  
  
 TreeNode(int value, TreeNode \*left, TreeNode \*right, TreeNode \*parent)  
 : Value(value), Left(left), Right(right), Parent(parent) {}  
 TreeNode() : Left(NULL), Right(NULL), Parent(NULL) {}  
 TreeNode(int value) : TreeNode(value, NULL, NULL, NULL) {}  
};  
  
void insert(TreeNode \*pParent, int value)  
{  
 if (pParent == NULL)  
 return;  
  
 if (value < pParent->Value)  
 {  
 if (pParent->Left == NULL)  
 pParent->Left = new TreeNode(value, NULL, NULL, pParent);  
 else  
 insert(pParent->Left, value);  
 }  
 else if (value > pParent->Value)  
 {  
 if (pParent->Right == NULL)  
 pParent->Right = new TreeNode(value, NULL, NULL, pParent);  
 else  
 insert(pParent->Right, value);  
 }  
}  
  
TreeNode \*search(TreeNode \*root, int value)  
{  
 if (root == NULL)  
 return NULL;  
 TreeNode \*pRoot = root;  
 while (pRoot != NULL)  
 {  
 if (value == pRoot->Value)  
 return pRoot;  
 else if (value < pRoot->Value)  
 pRoot = pRoot->Left;  
 else if (value > pRoot->Value)  
 pRoot = pRoot->Right;  
 }  
 return NULL;  
}  
  
void deleteTree(TreeNode \*\*root)  
{  
 TreeNode \*pRoot = \*root;  
 if (\*root == NULL)  
 return;  
 deleteTree(&(pRoot->Left));  
 deleteTree(&(pRoot->Right));  
 delete pRoot;  
 \*root = NULL;  
}  
  
void printTree(TreeNode \*root)  
{  
 if (root == NULL)  
 return;  
 printTree(root->Left);  
 cout << root->Value << " ";  
 printTree(root->Right);  
}  
  
TreeNode \*randomTree(uniform\_int\_distribution<int> &d, int size = 10)  
{  
 random\_device rd;  
 mt19937 generator(rd());  
 TreeNode \*root = new TreeNode(d(generator));  
 for (int i = 2; i <= size; i++)  
 insert(root, d(generator));  
  
 return root;  
}  
  
TreeNode \*create\_mirror\_flip(TreeNode \*pRoot)  
{  
 if (pRoot == NULL)  
 return NULL;  
  
 TreeNode \*pNewRoot = new TreeNode(pRoot->Value);  
 pNewRoot->Right = create\_mirror\_flip(pRoot->Left);  
 pNewRoot->Left = create\_mirror\_flip(pRoot->Right);  
  
 return pNewRoot;  
}  
  
void tree\_sum(TreeNode \*pRoot)  
{  
 if (pRoot == NULL)  
 return;  
  
 if (pRoot->Left == NULL && pRoot->Right == NULL)  
 return;  
  
 tree\_sum(pRoot->Left);  
 tree\_sum(pRoot->Right);  
  
 if (pRoot->Left != NULL || pRoot->Right != NULL)  
 {  
 pRoot->Value = 0;  
 if (pRoot->Left != nullptr)  
 pRoot->Value += pRoot->Left->Value;  
 if (pRoot->Right != nullptr)  
 pRoot->Value += pRoot->Right->Value;  
 }  
  
 return;  
}  
  
int main()  
{  
 uniform\_int\_distribution<int> dist(1, 100);  
  
 TreeNode \*root1 = randomTree(dist, 15);  
 cout << "Task 1\n";  
 cout << "Before flip: ";  
 printTree(root1);  
 cout << "\nAfter flip: ";  
 TreeNode \*flipped = create\_mirror\_flip(root1);  
 printTree(flipped);  
 // cout << "\nBefore flip: ";  
 // printTree(root1);  
  
 cout << "\nTask 2\n";  
 cout << "Sum: ";  
 tree\_sum(root1);  
 printTree(root1);  
  
 deleteTree(&root1);  
 deleteTree(&flipped);  
 return 0;  
}

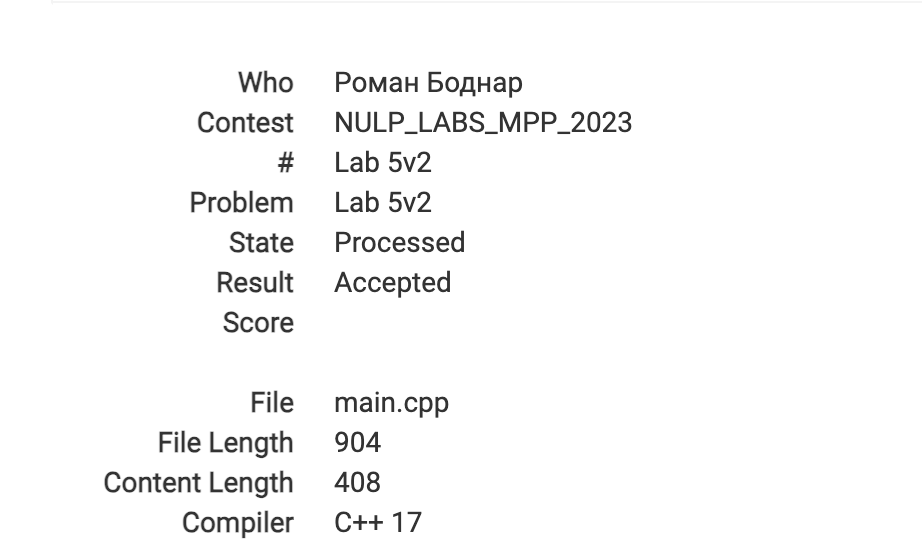
**4. Результати виконання завдань, тестування та фактично витрачений час:**

VNS Lab 10 – Variant 4:

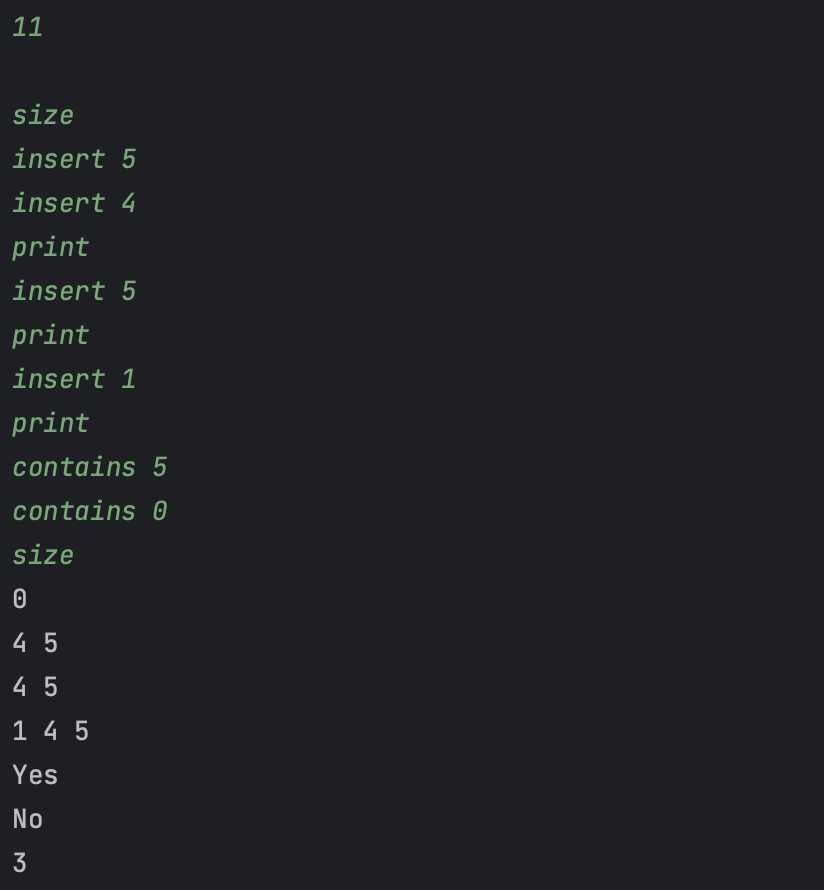


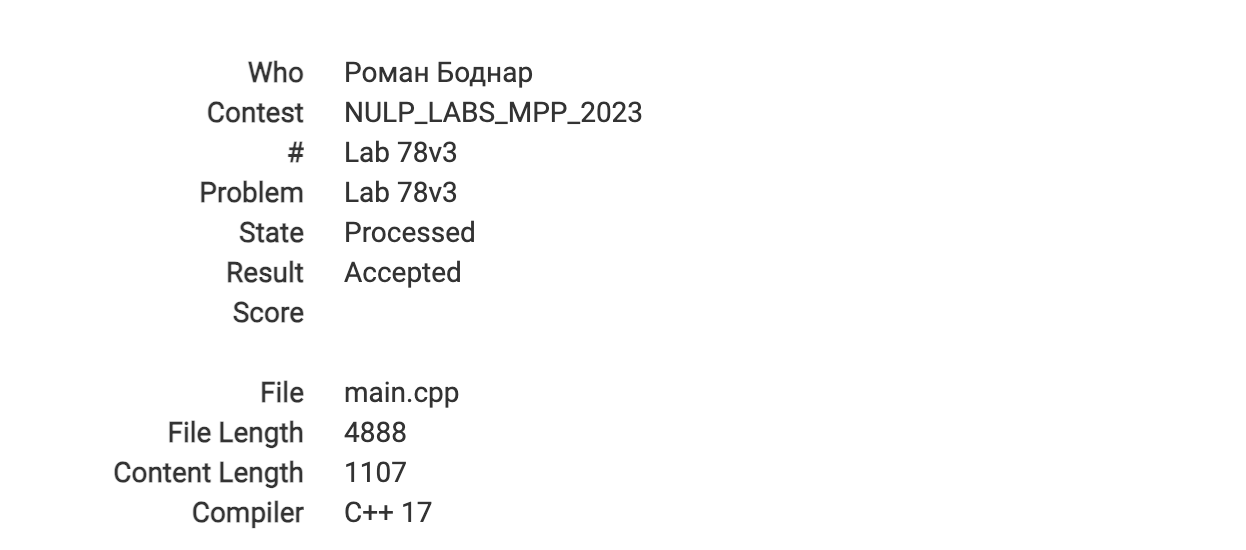
Algotester Lab 5v2:

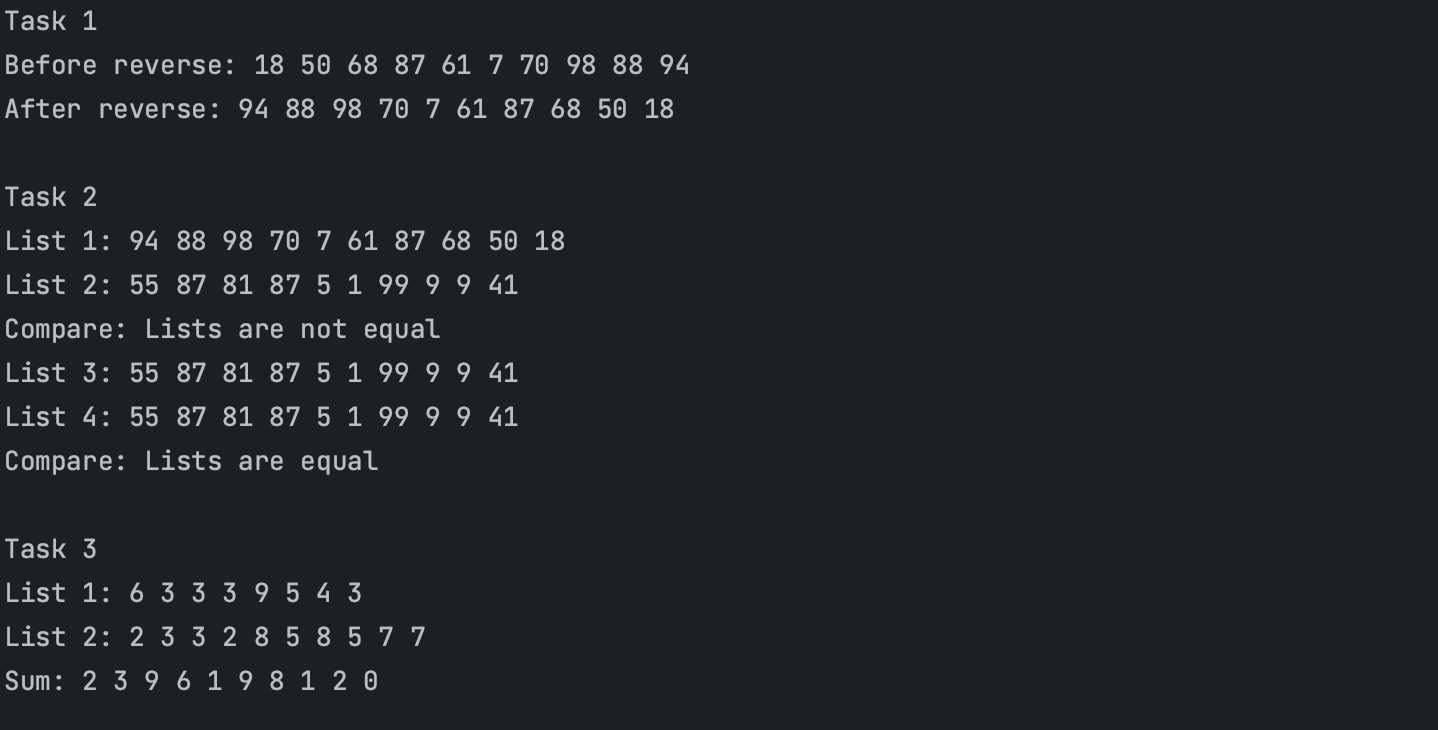


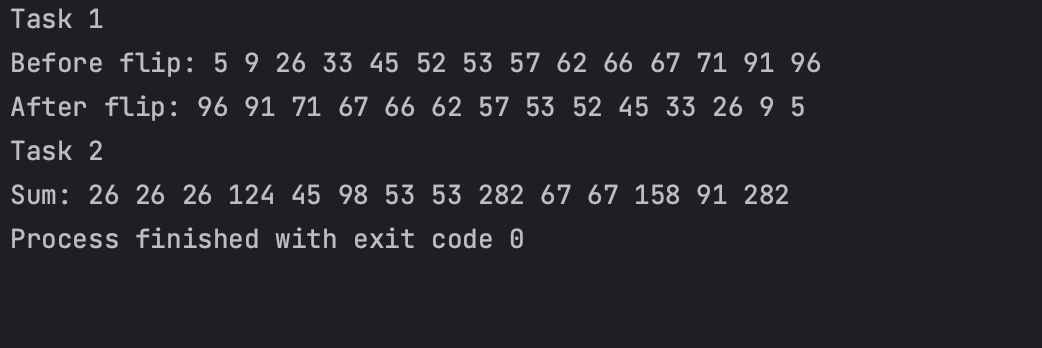


Algotester 78v3:





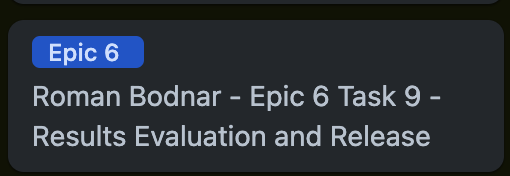
Class practice work:  
  


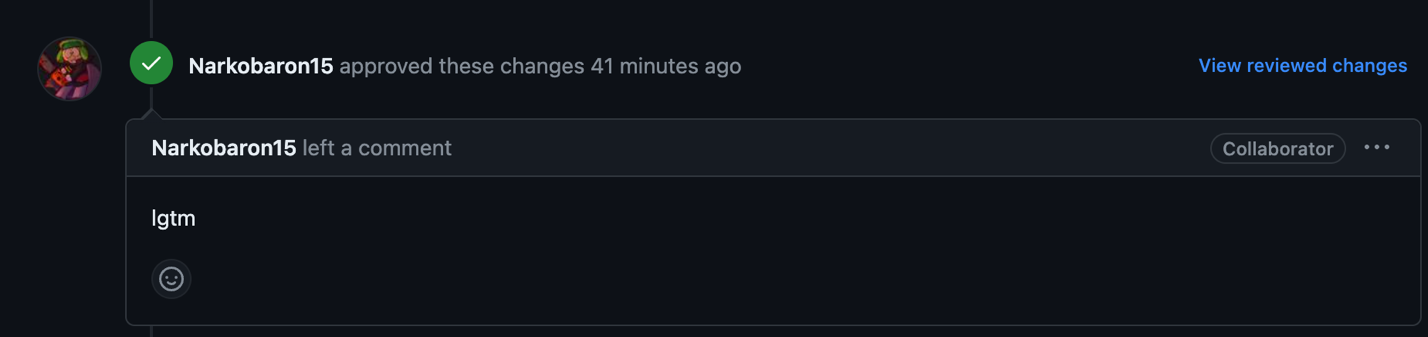


PR: <https://github.com/artificial-intelligence-department/ai_programming_playground/pull/1180>

**Скрін трелло по Епік 6:**





**Коментарі в PR Epic 6:**   


# Висновки:

В даній лабораторній роботі я вивчив теорію і застосування динамічних структур та алгоритмів обробки динамічних структур.