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

Іванов Олексій Олександрович

# **Тема роботи:**

Реалізація та використання динамічних структур даних**Мета роботи:**

* Ознайомитися з основними типами динамічних структур даних: стеком, чергою, списком, деревом.
* Вивчити основні операції над динамічними структурами даних.
* Закріпити навички реалізації алгоритмів з використанням колекцій на мові програмування С++.

# **Теоретичні відомості:**

1. Теоретичні відомості з переліком важливих тем:

* Тема №1: Список (однозв’язний, двозв’язний (в тому числі як нахилене дерево)).
* Тема №2: Черга (двостороння, одностороння (FIFO)) та стек (LIFO)
* Тема №3: Бінарне дерево пошуку (звичайне та збалансоване (AVL-збалансоване, червоно-чорне))

1. Індивідуальний план опрацювання теорії:

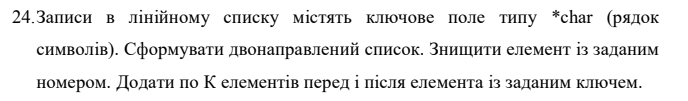
* Тема №3: Бінарні дерево пошуку.
  + Джерела Інформації:
    - https://www.geeksforgeeks.org/introduction-to-avl-tree/
    - https://www.geeksforgeeks.org/insertion-in-an-avl-tree/
    - https://www.geeksforgeeks.org/deletion-in-an-avl-tree/
    - https://www.programiz.com/dsa/avl-tree
    - https://www.geeksforgeeks.org/introduction-to-red-black-tree/
    - https://www.youtube.com/watch?v=DB1HFCEdLxA
    - https://www.youtube.com/watch?v=JPI-DPizQYk
    - https://www.youtube.com/watch?v=qvZGUFHWChY
  + Що опрацьовано:
    - Процес вставки/видалення для AVL дерева, ознайомлений з базовими принципами RBT, відмінностями та оптимальними випадками використання.
  + Статус: Ознайомлений
  + Початок опрацювання теми: 4/12/2023
  + Звершення опрацювання теми: 10/12/2023

# **Виконання роботи:**

## **1. Опрацювання завдання та вимог до програм та середовища:**

Завдання № 1 VNS Lab 10

* Варіант завдання - 24
* Деталі завдання

Рисунок 1: Деталі завдання VNS Lab 10

Завдання № 2 Class practice work

* Деталі завдання

**Задача №1 - Реверс списку (Reverse list)**

Реалізувати метод реверсу списку: Node\* reverse(Node \*head);

*Умови задачі:*

*- використовувати цілочисельні значення в списку;*

*- реалізувати метод реверсу;*

*- реалізувати допоміжний метод виведення вхідного і обернутого списків;*

**Задача №2 - Порівняння списків**

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

*Умови задачі:*

*- використовувати цілочисельні значення в списку;*

*- реалізувати функцію, яка ітеративно проходиться по обох списках і порівнює дані в кожному вузлі;*

*- якщо виявлено невідповідність даних або якщо довжина списків різна (один список закінчується раніше іншого), функція повертає false.*

**Задача №3 – Додавання великих чисел**

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

*Умови задачі:*

*- використовувати цифри від 0 до 9 для значень у списку;*

*- реалізувати функцію, яка обчислює суму двох чисел, які збережено в списку; молодший розряд числа записано в голові списка (напр. 379 ⟹ 9→7→3);*

*- функція повертає новий список, передані в функцію списки не модифікуються.*

**Задача №4 - Віддзеркалення дерева**

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

*Умови задачі:*

*- використовувати цілі числа для значень у вузлах дерева*

*- реалізувати функцію, що проходить по всіх вузлах дерева і міняє місцями праву і ліву вітки дерева*

*- функція повертає нове дерево, передане в функцію дерево не модифікується*

**Задача №5 - Записати кожному батьківському вузлу суму підвузлів**

void tree\_sum(TreeNode \*root);

*Умови задачі:*

*- використовувати цілочисельні значення у вузлах дерева;*

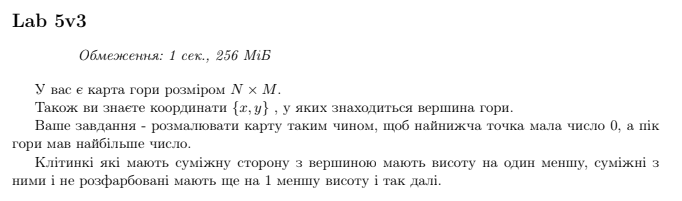
*- реалізувати функцію, яка ітеративно проходить по бінарному дереві і записує у батьківський вузол суму значень підвузлів*

*- вузол-листок не змінює значення*

*- значення змінюються від листків до кореня дерева*

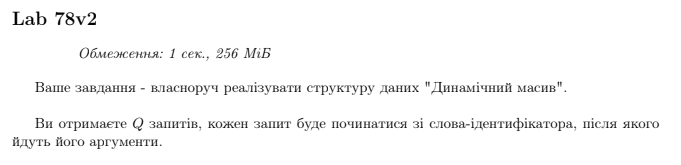
Завдання № 3 Algotester Lab 5

* Варіант завдання - 3
* Деталі завдання

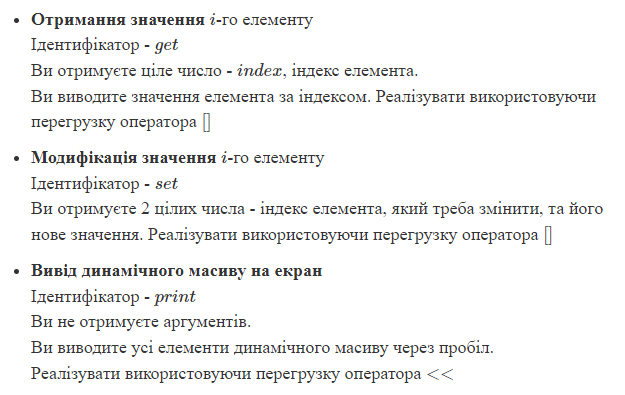
Рисунок 2: Деталі завдання Algotester Lab 5

Завдання № 4 Algotester Lab 7-8

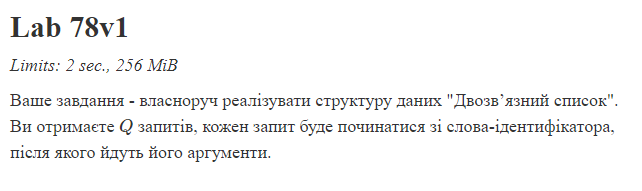
* Варіант завдання - 2
* [Деталі завдання](https://drive.google.com/file/d/1Fwcia22kDj7lRrTvjZYVhbuDtENM-XA1/view)
* Важливі деталі для врахування в імплементації програми

Рисунок 3: Деталі завдання Algotester Lab 7-8

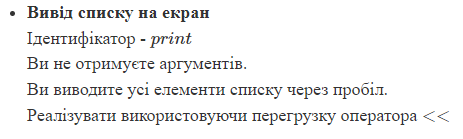
Завдання № 5 Self practice task 1

Рисунок 4: Важливі деталі в імплементації Algotester Lab 5

* [Деталі завдання](https://drive.google.com/file/d/1lVIPeC7NLGgECGo--hTg7fi9j4hqvd3S/view)

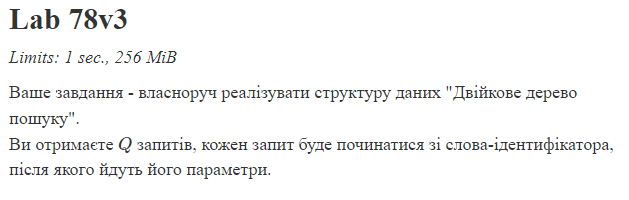
Рисунок 5: Деталі завдання Self practice task 1

* Важливі деталі для врахування в імплементації програми

Рисунок 6: Важливі деталі в імплементації Self practice task 1

Завдання № 6 Self practice task 2

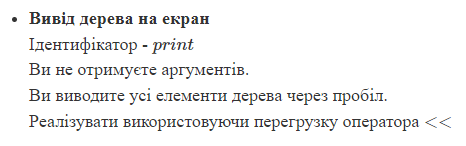
* [Деталі завдання](https://drive.google.com/file/d/1AzK1QACHUvJeJ8KXdQmMJ3C1x6oUq4XW/view)
* Важливі деталі для врахування в імплементації програми

Рисунок 7: Деталі завдання Self practice task 2

1) Self-constraint: make the AVL-balanced binary search tree

2)

## **2. Дизайн та планована оцінка часу виконання завдань:**

Рисунок 8: Важливі деталі в імплементації Self practice task 2

Програма № 1 "Інформаційні динамічні структури"

* Планований час на реалізацію — 30 хвилин
* Важливі деталі для врахування в імплементації

1) Використати С для написання програми

2) Використати файл для читання/запису даних  
 3) Реалізувати двозв’язний список для С-стрічок

Програма № 2 Зв’язаний список і бінарні дерева

* Планований час на реалізацію — 30 хвилин
* Важливі деталі для врахування в імплементації

Програма № 3 Карта висот

* [Блок-схема](https://github.com/artificial-intelligence-department/ai_programming_playground/pull/764/files" \l "diff-91ff7461e66c0b28e94c1b8f4d2f1e4ffdc8dce4d1b5dee36bc57ed27a5bcb73)
* Планований час на реалізацію — 20 хвилин
* Важливі деталі для врахування в імплементації

Варто вирахувати максимальну висоту до початку заповнення

Програма № 4 Динамічний масив (vector)

* Планований час на реалізацію — 45 хвилин
* Важливі деталі для врахування в імплементації

Зробити індексацію через оператор квадратних дужок (subscript operator)

Зробити виведення вектора через оператор <<

Програма № 5 Двозв’язний список

* Планований час на реалізацію — 1 година
* Важливі деталі для врахування в імплементації

Зробити індексацію через оператор квадратних дужок (subscript operator)

Зробити виведення вектора через оператор <<

Програма № 6 Двійкове дерево пошуку

* Планований час на реалізацію — 45 хвилин
* Важливі деталі для врахування в імплементації

Реалізувати збалансоване AVL дерево

## **3. Код програм з посиланням на зовнішні ресурси:**

Завдання № 1 [VNS Lab 10](https://github.com/artificial-intelligence-department/ai_programming_playground/pull/764/files" \l "diff-fad7bcd7546b38632ce7543d18412940742e5f175f37fbfe2d51f2a971b45b2c)

#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
// Doubly linked list  
  
typedef struct StrListNode {  
 char\* value;  
 struct StrListNode\* next;  
 struct StrListNode\* prev;  
} Node;  
  
Node\* create\_node(char\* value) {  
 Node\* node = (Node\*)malloc(sizeof(Node));  
 node->value = value;  
 node->next = node->prev = NULL;  
 return node;  
}  
  
void printList(Node\* head);  
  
void insert(Node\*\* head, char\* value, int index);  
void push\_front(Node\*\* head, char\* value);  
void push\_back(Node\*\* tail, char\* value);  
  
void remove\_elem(Node\*\* head, int index);  
void delete\_list(Node\*\* head);  
  
Node\* get\_elem(Node\* head, int index);  
Node\* get\_tail(Node\* head);  
  
int index\_of(Node\* head, char\* value);  
  
int get\_length(Node\* head);  
  
// End doubly linked list  
  
void insert\_k(Node\*\* head, char\* value, char\* key, int count);  
  
Node\* read\_from\_file(FILE\* f);  
void write\_to\_file(FILE\* f, Node\* head);  
Node\* seed();  
  
int main() {  
 FILE \*file = fopen("input.txt", "r");  
 Node \*head = NULL;  
  
 // seed the file with some data  
 if (file == NULL) head = seed();  
 else {  
 head = read\_from\_file(file);  
 fclose(file);  
 if (head == NULL) head = seed();  
 }  
  
 printList(head);  
  
 // Task 1  
 printf("Enter the index of the element to remove: ");  
 int index;  
 scanf\_s("%d", &index);  
 getc(stdin); // clear stdin (remove \n from buffer)  
 remove\_elem(&head, index);  
  
 // Task 2  
 char \*key = (char \*) calloc(sizeof(char), 256);  
 printf("Enter the key of element: ");  
 fgets(key, 256, stdin);  
  
 int count;  
 printf("Enter the count of elements to insert: ");  
 scanf\_s("%d", &count);  
 getc(stdin); // clear stdin (remove \n from buffer)  
  
 char \*value = (char \*) calloc(sizeof(char), 256);  
 printf("Enter the value of the element to insert: ");  
 fgets(value, 256, stdin);  
  
 insert\_k(&head, value, key, count);  
  
 printList(head);  
  
 file = fopen("input.txt", "w+");  
 write\_to\_file(file, head);  
 fclose(file);  
  
 delete\_list(&head);  
 return 0;  
}  
  
Node\* read\_from\_file(FILE\* f) {  
 char\* line;  
 int len = 256;  
  
 Node\* head = NULL, \*tail = NULL;  
  
 while (!feof(f)) {  
 line = (char\*)calloc(sizeof(char), len);  
 fgets(line, len, f);  
  
 if (line == NULL) break;  
  
 if (head == NULL) head = tail = create\_node(line);  
 else push\_back(&tail, line);  
 }  
  
 return head;  
}  
void write\_to\_file(FILE\* f, Node\* head) {  
 Node\* current = head;  
 while (current != NULL) {  
 fprintf(f, "%s", current->value);  
 if (current->value[strlen(current->value) - 1] != '\n')  
 fprintf(f, "\n");  
 current = current->next;  
 }  
}  
Node\* seed() {  
 FILE\* f = fopen("input.txt", "w+");  
 Node \*head = create\_node("Line 0");  
 Node \*tail = head;  
  
 for (int i = 1; i <= 10; ++i) {  
 char \*str = (char \*) malloc(sizeof(char) \* 10);  
 strcpy\_s(str, 10, "Line ");  
 char num[5];  
 itoa(i, num, 10);  
 strcat(str, num);  
 push\_back(&tail, str);  
 }  
  
 write\_to\_file(f, head);  
 fclose(f);  
 return head;  
}  
  
void insert\_k(Node\*\* head, char\* value, char\* key, int count) {  
 Node\* current = \*head;  
 int ind = index\_of(current, key);  
  
 if (ind == -1) {  
 perror("Key not found\n");  
 return;  
 }  
  
 for (int i = 0; i < count; ++i)  
 insert(head, value, ind + 1);  
 for (int i = 0; i < count; ++i)  
 insert(head, value, ind);  
}  
  
void printList(Node\* head) {  
 Node\* current = head;  
 if (current == NULL) {  
 printf("List is empty\n");  
 return;  
 }  
 while (current != NULL) {  
 printf("%s", current->value);  
 current = current->next;  
 }  
}  
  
int index\_of(Node\* head, char\* value) {  
 Node\* current = head;  
 int i = 0;  
 while (current != NULL) {  
 if (strcmp(current->value, value) == 0) return i;  
 current = current->next;  
 i++;  
 }  
 return -1;  
}  
  
Node\* get\_tail(Node\* head) {  
 Node\* current = head;  
 while (current->next != NULL) {  
 current = current->next;  
 }  
 return current;  
}  
  
void insert(Node\*\* head, char\* value, int index) {  
 Node\* current = \*head;  
 Node\* newNode = (Node\*)malloc(sizeof(Node));  
 newNode->value = value;  
 newNode->next = NULL;  
 newNode->prev = NULL;  
  
 if (index == 0) {  
 push\_front(head, value);  
 return;  
 }  
  
 int i = 0;  
 while (i < index - 1) {  
 current = current->next;  
 i++;  
 }  
  
 newNode->next = current->next;  
 newNode->prev = current;  
 current->next = newNode;  
}  
  
void push\_front(Node\*\* head, char\* value) {  
 Node\* newNode = (Node\*)malloc(sizeof(Node));  
 newNode->value = value;  
 newNode->next = \*head;  
 newNode->prev = NULL;  
 \*head = newNode;  
}  
  
void push\_back(Node\*\* tail, char\* value) {  
 Node\* newNode = create\_node(value);  
 (\*tail)->next = newNode;  
 newNode->value = value;  
 newNode->prev = \*tail;  
 \*tail = newNode;  
}  
  
void remove\_elem(Node\*\* head, int index) {  
 Node\* current = \*head;  
 if (index == 0) {  
 \*head = current->next;  
 free(current);  
 return;  
 }  
  
 int i = 0;  
 while (i < index - 1) {  
 current = current->next;  
 i++;  
 }  
  
 Node\* temp = current->next;  
 current->next = temp->next;  
 free(temp);  
}  
  
void delete\_list(Node\*\* head) {  
 Node\* current = \*head;  
 Node\* next = NULL;  
 while (current != NULL) {  
 next = current->next;  
 free(current);  
 current = next;  
 }  
 \*head = NULL;  
}  
  
Node\* get\_elem(Node\* head, int index) {  
 Node\* current = head;  
 int i = 0;  
 while (i < index) {  
 current = current->next;  
 i++;  
 }  
 return current;  
}  
  
int get\_length(Node\* head) {  
 Node\* current = head;  
 int length = 0;  
 while (current != NULL) {  
 length++;  
 current = current->next;  
 }  
 return length;  
}

Завдання № 2.1 [Class practice work (Linked lists)](https://github.com/artificial-intelligence-department/ai_programming_playground/pull/764/files" \l "diff-13caeb788f7c01f77b64e915e3dddb5b68115a7a3451c236bfc17425d7dfadc7)

#include <stdio.h>  
#include <stdlib.h>  
#include <stdbool.h>  
  
// Doubly linked list  
  
typedef struct IntListNode {  
 int value;  
 struct IntListNode\* next;  
 struct IntListNode\* prev;  
} Node;  
  
Node\* create\_node(int value) {  
 Node\* node = (Node\*)calloc(sizeof(Node), 1);  
 node->value = value;  
 return node;  
}  
  
void printList(Node\* head);  
  
void insert(Node\*\* head, int value, int index);  
void push\_front(Node\*\* head, int value);  
void push\_back(Node\*\* tail, int value);  
  
void remove\_elem(Node\*\* head, int index);  
void delete\_list(Node\*\* head);  
  
Node\* get\_elem(Node\* head, int index);  
Node\* get\_tail(Node\* head);  
  
int get\_length(Node\* head);  
  
// End doubly linked list  
  
// Task 1  
Node\* reverseList(Node\* head, Node\* tail);  
void Task1();  
  
// Task 2  
bool compare(Node \*h1, Node \*h2);  
void Task2();  
  
// Task 3  
Node\* add(Node\* h1, Node\* h2);  
void Task3();  
  
int main() {  
 Task1();  
 printf("\n");  
 Task2();  
 printf("\n");  
 Task3();  
  
 return 0;  
}  
void Task1() {  
 printf("Task 1:\n");  
 Node\* head = (Node\*)calloc(sizeof(Node), 1);  
 Node\* tail = head;  
  
 for (int i = 1; i < 10; i++) {  
 push\_back(&tail, i);  
 }  
 printf("Original list: ");  
 printList(head);  
  
 head = reverseList(head, tail);  
 printf("Reversed list: ");  
 printList(head);  
 delete\_list(&head);  
}  
  
void Task2() {  
 printf("Task 2:\n");  
  
 Node\* head1 = (Node\*)calloc(sizeof(Node), 1);  
 Node\* tail1 = head1;  
 for (int i = 1; i < 10; i++) {  
 push\_back(&tail1, i);  
 }  
  
 Node\* head2 = (Node\*)calloc(sizeof(Node), 1);  
 Node\* tail2 = head2;  
 for (int i = 1; i < 10; i++) {  
 push\_back(&tail2, i);  
 }  
  
 printList(head1);  
 printList(head2);  
 printf("Are they equal: %s\n\n", compare(head1, head2) ? "true" : "false");  
  
 head2 = reverseList(head2, tail2);  
 printList(head1);  
 printList(head2);  
 printf("Are they equal: %s\n", compare(head1, head2) ? "true" : "false");  
  
 delete\_list(&head1);  
 delete\_list(&head2);  
}  
  
void Task3() {  
 printf("Task 3:\n");  
  
 Node\* head1 = (Node\*)calloc(sizeof(Node), 1);  
 Node\* tail1 = head1;  
 head1->value = 9;  
 for (int i = 8; i > 6; i--) {  
 push\_back(&tail1, i);  
 }  
  
 Node\* head2 = (Node\*)calloc(sizeof(Node), 1);  
 Node\* tail2 = head2;  
 head2->value = 7;  
 for (int i = 8; i > 3; i--) {  
 push\_back(&tail2, i);  
 }  
  
 printf("First number: ");  
 printList(head1);  
 printf("Second number: ");  
 printList(head2);  
  
 Node\* result = add(head1, head2);  
 printf("Result (a + b): ");  
 printList(result);  
  
 delete\_list(&head1);  
 delete\_list(&head2);  
 delete\_list(&result);  
}  
  
Node\* reverseList(Node\* head, Node\* tail) {  
 Node\* current = head;  
 Node\* temp = NULL;  
 while (current != NULL) {  
 // swap next and prev pointers  
 temp = current->next;  
 current->next = current->prev;  
 current->prev = temp;  
  
 // move to next node  
 current = temp;  
 }  
  
 // return new head  
 return tail;  
}  
  
bool compare(Node \*h1, Node \*h2) {  
 Node\* current1 = h1;  
 Node\* current2 = h2;  
  
 while (current1 != NULL && current2 != NULL) {  
 if (current1->value != current2->value) {  
 return false;  
 }  
 current1 = current1->next;  
 current2 = current2->next;  
 }  
 if (current1 != NULL || current2 != NULL) {  
 return false;  
 }  
 return true;  
}  
  
Node\* add(Node\* h1, Node\* h2) {  
 Node\* current1 = get\_tail(h1);  
 Node\* current2 = get\_tail(h2);  
 Node\* resultHead = (Node\*)calloc(sizeof(Node), 1);  
  
 int carry = 0;  
  
 while (current1 != NULL && current2 != NULL) {  
 int sum = current1->value + current2->value + carry;  
 carry = sum / 10;  
 sum = sum % 10;  
 push\_front(&resultHead, sum);  
 current1 = current1->prev;  
 current2 = current2->prev;  
 }  
 while (current1 != NULL) {  
 int sum = current1->value + carry;  
 carry = sum / 10;  
 sum = sum % 10;  
 push\_front(&resultHead, sum);  
 current1 = current1->prev;  
 }  
 while (current2 != NULL) {  
 int sum = current2->value + carry;  
 carry = sum / 10;  
 sum = sum % 10;  
 push\_front(&resultHead, sum);  
 current2 = current2->prev;  
 }  
  
 if (carry != 0) {  
 push\_front(&resultHead, carry);  
 }  
  
 remove\_elem(&resultHead, get\_length(resultHead) - 1);  
 return resultHead;  
}  
  
void printList(Node\* head) {  
 Node\* current = head;  
 while (current != NULL) {  
 printf("%d ", current->value);  
 current = current->next;  
 }  
 printf("\n");  
}  
  
Node\* get\_tail(Node\* head) {  
 Node\* current = head;  
 while (current->next != NULL) {  
 current = current->next;  
 }  
 return current;  
}  
  
void insert(Node\*\* head, int value, int index) {  
 Node\* current = \*head;  
 Node\* newNode = (Node\*)malloc(sizeof(Node));  
 newNode->value = value;  
 newNode->next = NULL;  
 newNode->prev = NULL;  
  
 if (index == 0) {  
 push\_front(head, value);  
 return;  
 }  
  
 int i = 0;  
 while (i < index - 1) {  
 current = current->next;  
 i++;  
 }  
  
 newNode->next = current->next;  
 newNode->prev = current;  
 current->next = newNode;  
}  
  
void push\_front(Node\*\* head, int value) {  
 Node\* newNode = (Node\*)malloc(sizeof(Node));  
 newNode->value = value;  
 newNode->next = \*head;  
 newNode->prev = NULL;  
 \*head = newNode;  
}  
  
void push\_back(Node\*\* tail, int value) {  
 Node\* newNode = (Node\*)malloc(sizeof(Node));  
 (\*tail)->next = newNode;  
 newNode->value = value;  
 newNode->next = NULL;  
 newNode->prev = \*tail;  
 \*tail = newNode;  
}  
  
void remove\_elem(Node\*\* head, int index) {  
 Node\* current = \*head;  
 if (index == 0) {  
 \*head = current->next;  
 free(current);  
 return;  
 }  
  
 int i = 0;  
 while (i < index - 1) {  
 current = current->next;  
 i++;  
 }  
  
 Node\* temp = current->next;  
 current->next = temp->next;  
 free(temp);  
}  
  
void delete\_list(Node\*\* head) {  
 Node\* current = \*head;  
 Node\* next = NULL;  
 while (current != NULL) {  
 next = current->next;  
 free(current);  
 current = next;  
 }  
 \*head = NULL;  
}  
  
Node\* get\_elem(Node\* head, int index) {  
 Node\* current = head;  
 int i = 0;  
 while (i < index) {  
 current = current->next;  
 i++;  
 }  
 return current;  
}  
  
int get\_length(Node\* head) {  
 Node\* current = head;  
 int length = 0;  
 while (current != NULL) {  
 length++;  
 current = current->next;  
 }  
 return length;  
}

Завдання № 2.2 [Class practice work (Binary trees)](https://github.com/artificial-intelligence-department/ai_programming_playground/pull/764/files" \l "diff-c24168c00b68c6ba7ef500cf1599b671a7b57dda3126f0c3486b481b83d34898)

#include <stdio.h>  
#include <stdlib.h>  
  
// BST - binary search tree  
typedef struct BSTNode {  
 int value;  
  
 struct BSTNode\* parent;  
 struct BSTNode\* left;  
 struct BSTNode\* right;  
} TreeNode;  
  
TreeNode\* create\_bst\_root(int value);  
TreeNode\* create\_node(int value, TreeNode\* parent);  
void emplace(TreeNode\* root, int value);  
void delete\_tree(TreeNode\* root);  
// In order traversal printing  
void inorder\_print(TreeNode\* root);  
// End BST  
  
// Task 4  
TreeNode\* create\_mirror\_flip(TreeNode \*node);  
void Task4(TreeNode \*root);  
  
// Task 5  
TreeNode\* tree\_sum(TreeNode \*root); // I made it a safe procedure  
void Task5(TreeNode \*root);  
  
int main() {  
 int values[] = {7, 3, 11, 1, 5, 9, 13, 0, 2, 4, 6, 8, 10, 12, 14}, n = 15;  
  
 TreeNode \*root = create\_bst\_root(values[0]);  
 for (int i = 1; i < n; i++) emplace(root, values[i]);  
  
 Task4(root);  
 printf("\n");  
 Task5(root);  
  
 delete\_tree(root);  
 return 0;  
}  
  
void Task4(TreeNode \*root) {  
 printf("Task 4:\n");  
  
 printf("Original tree: ");  
 inorder\_print(root);  
  
 printf("\nMirror tree: ");  
 TreeNode \*mirror = create\_mirror\_flip(root);  
 inorder\_print(mirror);  
  
 delete\_tree(mirror);  
}  
  
void Task5(TreeNode \*root) {  
 printf("Task 5:\n");  
 printf("Original tree: ");  
 inorder\_print(root);  
 printf("\nTree sum: ");  
 TreeNode \*sum = tree\_sum(root);  
 inorder\_print(sum);  
  
 delete\_tree(sum);  
}  
  
TreeNode\* create\_bst\_root(int value) {  
 return create\_node(value, NULL);  
}  
  
TreeNode\* create\_node(int value, TreeNode\* parent) {  
 TreeNode\* node = (TreeNode\*)malloc(sizeof(TreeNode));  
 node->left = node->right = NULL;  
 node->value = value;  
 node->parent = parent;  
 return node;  
}  
  
void emplace(TreeNode\* root, int value) {  
 if (root->value == value) return;  
  
 if (value < root->value) {  
 if (root->left == NULL) root->left = create\_node(value, root);  
 else emplace(root->left, value);  
 } else {  
 if (root->right == NULL) root->right = create\_node(value, root);  
 else emplace(root->right, value);  
 }  
}  
  
void delete\_tree(TreeNode\* root) {  
 if (root == NULL) return;  
  
 delete\_tree(root->left);  
 delete\_tree(root->right);  
 free(root);  
}  
  
void inorder\_print(TreeNode\* root) {  
 if (root == NULL) return;  
  
 inorder\_print(root->left);  
 printf("%d ", root->value);  
 inorder\_print(root->right);  
}  
  
TreeNode \*create\_mirror\_flip(TreeNode \*node) {  
 if (node == NULL) return NULL;  
  
 TreeNode \*mirror = create\_node(node->value, node->parent);  
 if (node->right) mirror->left = create\_mirror\_flip(node->right);  
 if (node->left) mirror->right = create\_mirror\_flip(node->left);  
  
 return mirror;  
}  
  
TreeNode\* tree\_sum(TreeNode \*root) {  
 if (root == NULL) return NULL;  
  
 TreeNode \*sum = create\_node(root->value, NULL);  
 sum->left = tree\_sum(root->left);  
 sum->right = tree\_sum(root->right);  
  
 if (sum->left == NULL && sum->right == NULL) return sum;  
  
 sum->value = 0;  
 if (sum->left != NULL) sum->value += sum->left->value;  
 if (sum->right != NULL) sum->value += sum->right->value;  
  
 return sum;  
}

Завдання № 3 [Algotester Lab 5](https://github.com/artificial-intelligence-department/ai_programming_playground/pull/764/files" \l "diff-447b00d0e311a6f5eedcdcb37a1f5bb83e8bf4689ef849626e45739732622ed9)

#include <iostream>  
#include <vector>  
using namespace std;  
  
// check if the point (x, y) is within the map  
bool check\_offsets(int x, int y, int m, int p) {  
 return x >= 0 && x < m && y >= 0 && y < p;  
}  
  
void fill\_in\_from\_pinnacle(vector<vector<int>>& map, int x, int y) {  
 int n = map.size(), m = map[0].size();  
 int max\_by\_x = (x > n - x) ? x - 1 : n - x,  
 max\_by\_y = (y > m - y) ? y - 1 : m - y;  
  
 int max = max\_by\_x + max\_by\_y;  
  
 map[--x][--y] = max;  
  
 // fill in the map starting from the pinnacle  
 for(int i = 0; i <= max\_by\_x; i++) {  
 for (int j = 0; j <= max\_by\_y; j++) {  
 // top left  
 if (check\_offsets(x - i, y - j, n, m))  
 map[x - i][y - j] = max - i - j;  
 // top right  
 if (check\_offsets(x - i, y + j, n, m))  
 map[x - i][y + j] = max - i - j;  
 // bottom left  
 if (check\_offsets(x + i, y - j, n, m))  
 map[x + i][y - j] = max - i - j;  
 // bottom right  
 if (check\_offsets(x + i, y + j, n, m))  
 map[x + i][y + j] = max - i - j;  
 }  
 }  
}  
  
int main() {  
 int n, m;  
 cin >> n >> m;  
  
 vector<vector<int>> map(n, vector<int>(m, 0));  
  
 // Point of the pinnacle  
 int x, y;  
 cin >> x >> y;  
  
 fill\_in\_from\_pinnacle(map, x, y);  
  
 for (const vector<int> &row: map) {  
 for (int cell: row)  
 cout << cell << " ";  
 cout << endl;  
 }  
  
 return 0;  
}

Завдання № 4 [Algotester Lab 7-8](https://github.com/artificial-intelligence-department/ai_programming_playground/pull/764/files" \l "diff-1ad1f5c0d72397ad75bf0961d0e96418db2819f28a500cd942541a09b561511b)

#include <iostream>  
#include <functional>  
  
// Algotester Lab 7-8 V2  
// Vector  
  
// TODO: Add iterators  
template<class T>  
class Vector {  
private:  
 static constexpr int DEFAULT\_CAPACITY = 1;  
 static constexpr int GROWTH\_FACTOR = 2;  
  
 T\* data;  
 int \_size;  
 int \_capacity;  
  
 void check\_capacity() {  
 if (\_size == \_capacity)  
 resize(\_capacity \* GROWTH\_FACTOR);  
 }  
  
 void check\_index(const int index) const {  
 if (index < 0 || index > \_size)  
 throw std::out\_of\_range("Index out of range");  
 }  
  
public:  
 explicit Vector(const int capacity) {  
 data = new T[capacity];  
 \_size = 0;  
 \_capacity = capacity;  
 }  
  
 Vector() : Vector(DEFAULT\_CAPACITY) {}  
  
 // If we use this ctor, it'll disable IDE warnings  
 // (CLion is not that smart tho)  
  
 // Vector() {  
 // data = new T[DEFAULT\_CAPACITY];  
 // \_size = 0;  
 // \_capacity = DEFAULT\_CAPACITY;  
 // }  
  
 Vector(std::initializer\_list<T> list)  
 : Vector(list.size()) {  
 for (auto& value : list) {  
 push\_back(value);  
 }  
 }  
  
 ~Vector() {  
 \_size = 0;  
 delete[] data;  
 }  
  
 void resize(const int new\_capacity) {  
 T\* new\_data = new T[new\_capacity];  
 for (int i = 0; i < \_size; i++) {  
 new\_data[i] = data[i];  
 }  
 delete[] data;  
 data = new\_data;  
 \_capacity = new\_capacity;  
 }  
  
 void insert(T value, int index) {  
 check\_capacity();  
 check\_index(index);  
  
 for (int i = \_size; i > index; i--) {  
 data[i] = data[i - 1];  
 }  
 data[index] = value;  
 \_size++;  
  
 check\_capacity();  
 }  
  
 void push\_back(T value) {  
 insert(value, \_size);  
 }  
  
 // Nodiscard is used to warn the user if the return value is ignored  
 // (Function is useless if we just call it)  
 [[nodiscard]] int size() const {  
 return \_size;  
 }  
  
 [[nodiscard]] bool empty() const {  
 return \_size == 0;  
 }  
  
 [[nodiscard]] int capacity() const {  
 return \_capacity;  
 }  
  
 void remove(const int index) {  
 check\_index(index);  
 for (int i = index; i < \_size - 1; i++) {  
 data[i] = data[i + 1];  
 }  
 \_size--;  
 }  
  
 void pop\_back() {  
 remove(\_size - 1);  
 }  
  
 void foreach(std::function<void(T)> func) const {  
 for (int i = 0; i < \_size; i++) {  
 func(data[i]);  
 }  
 }  
  
 T& operator[](int index) {  
 check\_index(index);  
 return data[index];  
 }  
};  
  
template<class T>  
std::ostream& operator<<(std::ostream& os, const Vector<T>& v) {  
 v.foreach([&os](T value) {  
 os << value << " ";  
 });  
 return os;  
}  
  
int main() {  
 using namespace std;  
 Vector<int> v;  
  
 int queries;  
 string query;  
 cin >> queries;  
  
 for (int i = 0; i < queries; ++i) {  
 cin >> query;  
  
 try {  
 if (query == "insert") {  
 int index, count;  
 cin >> index >> count;  
  
 vector<int> values(count);  
 for (int j = 0; j < count; ++j)  
 cin >> values[j];  
  
 for (int j = count - 1; j >= 0; --j)  
 v.insert(values[j], index);  
 } else if (query == "erase") {  
 int index, count;  
 cin >> index >> count;  
  
 for (int j = 0; j < count; ++j)  
 v.remove(index);  
 } else if (query == "size") {  
 cout << v.size() << endl;  
 } else if (query == "get") {  
 int index;  
 cin >> index;  
 cout << v[index] << endl;  
 } else if (query == "set") {  
 int index, value;  
 cin >> index >> value;  
 v[index] = value;  
 } else if (query == "print") {  
 cout << v << endl;  
 } else if (query == "capacity") {  
 cout << v.capacity() << endl;  
 }  
 }  
 // ignore out of range exceptions  
 catch (out\_of\_range&) {}  
 catch (exception&) {}  
 }  
}

Завдання № 5 [Self practice task 1](https://github.com/artificial-intelligence-department/ai_programming_playground/pull/764/files" \l "diff-c44a99fa1f371aeada77d9a68ca2f6f061922ca5e9d28f1db233881472c96b9e)

#include <iostream>  
#include <functional>  
  
// Algotester Lab 7-8 V1  
// Doubly linked list  
  
// *TODO - make this an iterable collection*template<class T>  
class DoublyLinkedList {  
protected:  
 struct Node {  
 public:  
 T value;  
 Node\* next;  
 Node\* prev;  
  
 Node() : next(nullptr), prev(nullptr) {}  
  
 Node(T value, Node\* next, Node\* prev)  
 : value(value), next(next), prev(prev) {}  
  
 explicit Node(T value)  
 : Node(value, nullptr, nullptr) {}  
 };  
  
 Node\* get\_node(const int index) {  
 check\_index(index);  
  
 const int mid = \_size / 2;  
 Node\* node;  
  
 if (index <= mid) {  
 node = head;  
 for (int i = 0; i < index; ++i) {  
 node = node->next;  
 }  
 } else {  
 node = tail;  
 for (int i = \_size - 1; i > index; --i) {  
 node = node->prev;  
 }  
 }  
  
 return node;  
 }  
  
 void check\_index(const int index) const {  
 if (index < 0 || index > \_size) {  
 throw std::out\_of\_range("Index out of range");  
 }  
 }  
  
 void create\_head(T value) {  
 head = tail = new Node(value);  
 \_size = 1;  
 }  
  
private:  
 Node\* head;  
 Node\* tail;  
 int \_size;  
  
public:  
 DoublyLinkedList() : head(nullptr), tail(nullptr), \_size(0) {}  
 DoublyLinkedList(std::initializer\_list<T> list)  
 : DoublyLinkedList() {  
 for (auto& value : list) {  
 push\_back(value);  
 }  
 }  
  
 ~DoublyLinkedList() {  
 while (head != nullptr) {  
 Node\* next = head->next;  
 delete head;  
 head = next;  
 }  
 \_size = 0;  
 }  
  
 // Nodiscard is used to warn the user if the return value is ignored  
 // (Function is useless if we just call it)  
 [[nodiscard]] int size() const {  
 return \_size;  
 }  
  
 [[nodiscard]] bool empty() const {  
 return \_size == 0;  
 }  
  
 int index\_of(T& element) {  
 Node\* node = head;  
 int index = 0;  
 while (node != nullptr) {  
 if (node->value == element) {  
 return index;  
 }  
 node = node->next;  
 index++;  
 }  
 return -1;  
 }  
  
 void push\_front(T value) {  
 if (head == nullptr) {  
 create\_head(value);  
 return;  
 }  
  
 Node\* node = new Node(value, head, nullptr);  
 head->prev = node;  
 head = node;  
 \_size++;  
 }  
  
 void push\_back(T value) {  
 if (head == nullptr) {  
 create\_head(value);  
 return;  
 }  
  
 Node\* node = new Node(value, nullptr, tail);  
 tail->next = node;  
 tail = node;  
 \_size++;  
 }  
  
 void insert(T value, const int index) {  
 check\_index(index);  
  
 if (index == 0) {  
 push\_front(value);  
 return;  
 }  
  
 if (index == \_size) {  
 push\_back(value);  
 return;  
 }  
  
 Node\* node = get\_node(index);  
 Node\* new\_node = new Node(value, node, node->prev);  
 node->prev->next = new\_node;  
 node->prev = new\_node;  
 \_size++;  
 }  
  
 void pop\_front() {  
 if (head == nullptr) return;  
  
 const Node\* node = head;  
 head = head->next;  
 delete node;  
 \_size--;  
  
 if (head == nullptr) tail = nullptr;  
 else head->prev = nullptr;  
 }  
  
 void pop\_back() {  
 if (tail == nullptr) return;  
  
 const Node\* node = tail;  
 tail = tail->prev;  
 delete node;  
 \_size--;  
  
 if (tail == nullptr) head = nullptr;  
 else tail->next = nullptr;  
 }  
  
 void erase(const int index) {  
 check\_index(index);  
  
 if (index == 0) {  
 pop\_front();  
 return;  
 }  
  
 if (index == \_size - 1) {  
 pop\_back();  
 return;  
 }  
  
 Node\* node = get\_node(index);  
 node->prev->next = node->next;  
 node->next->prev = node->prev;  
 delete node;  
 \_size--;  
 }  
  
 void remove(T& element) {  
 erase(index\_of(element));  
 }  
  
 void forEach(std::function<void(T&)> callback) {  
 Node\* node = head;  
 while (node != nullptr) {  
 callback(node->value);  
 node = node->next;  
 }  
 }  
  
 T& operator[](const int index) {  
 return get\_node(index)->value;  
 }  
};  
  
template<class T>  
std::ostream& operator<<(std::ostream& os, DoublyLinkedList<T>& list) {  
 list.forEach([&os](T& value) {  
 os << value << " ";  
 });  
 return os;  
}  
  
int main() {  
 using namespace std;  
 DoublyLinkedList<int> list;  
  
 int queries;  
 string query;  
 cin >> queries;  
  
 for (int i = 0; i < queries; ++i) {  
 cin >> query;  
  
 try {  
 if (query == "insert") {  
 int index, count;  
 cin >> index >> count;  
  
 DoublyLinkedList<int> values;  
 for (int j = 0; j < count; ++j) {  
 int value;  
 cin >> value;  
 values.push\_back(value);  
 }  
  
 for (int j = count - 1; j >= 0; --j)  
 list.insert(values[j], index);  
 } else if (query == "erase") {  
 int index, count;  
 cin >> index >> count;  
  
 for (int j = 0; j < count; ++j)  
 list.erase(index);  
 } else if (query == "size") {  
 cout << list.size() << endl;  
 } else if (query == "get") {  
 int index;  
 cin >> index;  
 cout << list[index] << endl;  
 } else if (query == "set") {  
 int index, value;  
 cin >> index >> value;  
 list[index] = value;  
 } else if (query == "print") {  
 cout << list << endl;  
 }  
 }  
 catch (out\_of\_range&) {}  
 catch (exception&) {}  
 }  
  
 return 0;  
}

Завдання № 6 [Self practice task 2](https://github.com/artificial-intelligence-department/ai_programming_playground/pull/764/files" \l "diff-eeba76da4611708dac42ec0caa9b9061a36aaa33161b83a8546e1c0be38b7678)

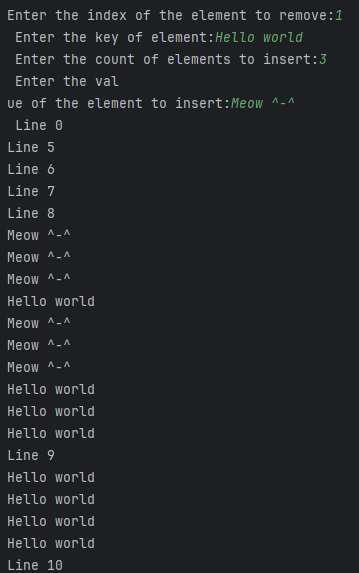
#include <iostream>  
#include <functional>  
  
// Algotester Lab 7-8 V3  
// Binary Search Tree (AVL)  
  
// TODO: Add iterators  
template <class T>  
class AVL {  
protected:  
 struct Node {  
 public:  
 T value;  
 int height;  
 // Node\* parent;  
 Node\* left;  
 Node\* right;  
  
 Node(T value, int height, Node\* left, Node\* right)  
 : value(value), height(height), left(left), right(right) {}  
  
 Node() : height(0), left(nullptr), right(nullptr) {}  
  
 explicit Node(T value) : Node(value, 0, nullptr, nullptr) {}  
 };  
  
 // Helper for finalizer  
 void delete\_tree(Node\* \_root) {  
 if (\_root == nullptr) return;  
  
 delete\_tree(\_root->left);  
 delete\_tree(\_root->right);  
 delete \_root;  
 }  
  
 Node\* find(T value) {  
 Node\* current = root;  
 while (current != nullptr) {  
 if (current->value == value) return current;  
 if (value < current->value) current = current->left;  
 else current = current->right;  
 }  
 return nullptr;  
 }  
  
 Node\* \_append\_node(Node\* node, T value) {  
 if (node == nullptr) {  
 \_size++;  
 return new Node(value);  
 }  
  
 // traverse the tree to find the right place for the new node  
 if (value < node->value)  
 node->left = \_append\_node(node->left, value);  
 else if (value > node->value)  
 node->right = \_append\_node(node->right, value);  
  
 // Balance the tree  
  
 // Update height of the root  
 const int left\_height = height(node->left),  
 right\_height = height(node->right);  
 // determine height of root (max of left or right subtree + 1)  
 // (because it is the longest path from root to leaf)  
 node->height = 1 + std::max(left\_height, right\_height);  
  
 // determine balance factor  
 const int balance\_factor = left\_height - right\_height;  
  
 // if balance\_factor is greater than 1  
 // then tree is left heavy -> right rotation  
 if (balance\_factor > 1) {  
 if (value < node->left->value) {  
 return right\_rotation(node);  
 } else {  
 // left rotation on left subtree (because it is right heavy)  
 node->left = left\_rotation(node->left);  
 return right\_rotation(node);  
 }  
 }  
 // if balance\_factor is less than -1  
 // then tree is right heavy -> left rotation  
 else if (balance\_factor < -1) {  
 if (value > node->right->value) {  
 return left\_rotation(node);  
 } else {  
 // right rotation on right subtree (because it is left heavy)  
 node->right = right\_rotation(node->right);  
 return left\_rotation(node);  
 }  
 }  
 return node;  
 }  
  
 Node\* \_remove\_node(Node\* node, T value) {  
 if (node == nullptr) return nullptr;  
  
 // traverse the tree to find the right node  
 if (value < node->value) {  
 node->left = \_remove\_node(node->left, value);  
 } else if (value > node->value) {  
 node->right = \_remove\_node(node->right, value);  
 } else /\* if reached the value \*/ {  
 Node \*r = node->right;  
 if (node->right == nullptr) {  
 Node \*l = node->left;  
 delete node;  
 node = l;  
 } else if (node->left == nullptr) {  
 delete node;  
 node = r;  
 } else /\* if both leaves are present \*/ {  
 // find the smallest value in the right subtree  
 while (r->left != nullptr) r = r->left;  
  
 // and replace the root with it  
 node->value = r->value;  
 node->right = \_remove\_node(node->right, r->value);  
 }  
 }  
  
 if (node == nullptr) return nullptr;  
  
 // Balance the tree  
  
 // Update height of the root  
 const int left\_height = height(node->left),  
 right\_height = height(node->right);  
 // determine height of root (max of left or right subtree + 1)  
 // (because it is the longest path from root to leaf)  
 node->height = 1 + std::max(left\_height, right\_height);  
  
 // determine balance factor  
 const int balance\_factor = left\_height - right\_height;  
 if (balance\_factor > 1) {  
 if (height(node->left) >= height(node->right))  
 return right\_rotation(node);  
 else {  
 node->left = left\_rotation(node->left);  
 return right\_rotation(node);  
 }  
 } else if (balance\_factor < -1) {  
 if (height(node->right) >= height(node->left))  
 return left\_rotation(node);  
 else {  
 node->right = right\_rotation(node->right);  
 return left\_rotation(node);  
 }  
 }  
 return node;  
 }  
  
 static size\_t height(const Node\* node) {  
 return node == nullptr ? 0 : node->height;  
 }  
  
 Node\* right\_rotation(Node\* \_root) {  
 // new root is the left child of the old root  
 // (anti-clockwise rotation)  
 Node \*new\_root = \_root->left;  
  
 \_root->left = new\_root->right;  
 // new root's right child is the old root  
 new\_root->right = \_root;  
  
 // update heights  
 \_root->height = 1 + std::max(  
 height(\_root->left),  
 height(\_root->right)  
 );  
 new\_root->height = 1 + std::max(  
 height(new\_root->left),  
 height(new\_root->right)  
 );  
  
 return new\_root;  
 }  
  
 Node\* left\_rotation(Node\* \_root){  
 // new root is the right child of the old root  
 // (clockwise rotation)  
 Node\* new\_root = \_root->right;  
  
 \_root->right = new\_root->left;  
 // new root's left child is the old root  
 new\_root->left = \_root;  
  
 // update heights  
 \_root->height = 1 + std::max(  
 height(\_root->left),  
 height(\_root->right)  
 );  
 new\_root->height = 1 + std::max(  
 height(new\_root->left),  
 height(new\_root->right)  
 );  
  
 return new\_root;  
 }  
  
private:  
 Node\* root;  
 int \_size;  
  
public:  
 enum Traversal {  
 PREORDER,  
 INORDER,  
 POSTORDER  
 };  
  
 AVL() {  
 root = nullptr;  
 \_size = 0;  
 }  
  
 AVL(std::initializer\_list<T> list) {  
 root = nullptr;  
 \_size = 0;  
 for (auto value : list) {  
 emplace(value);  
 }  
 }  
  
 ~AVL() {  
 delete\_tree(root);  
 }  
  
 [[nodiscard]] int size() const {  
 return \_size;  
 }  
  
 void emplace(T value) {  
 root = \_append\_node(root, value);  
 }  
  
 bool contains(T value) {  
 return find(value) != nullptr;  
 }  
  
 void remove(T value) {  
 root = \_remove\_node(root, value);  
 }  
  
 void forEach(std::function<void(T)> func, Traversal traversal = Traversal::INORDER) {  
 std::function<void(Node\*)> \_func = [&](Node\* node) {  
 if (node == nullptr) return;  
 if (traversal == Traversal::PREORDER) func(node->value);  
 \_func(node->left);  
 if (traversal == Traversal::INORDER) func(node->value);  
 \_func(node->right);  
 if (traversal == Traversal::POSTORDER) func(node->value);  
 };  
 \_func(root);  
 }  
};  
  
template <class T>  
std::ostream& operator<<(std::ostream& os, AVL<T>& \_tree) {  
 \_tree.forEach([&](T value) {  
 os << value << " ";  
 }, AVL<T>::Traversal::INORDER);  
 return os;  
}  
  
int main() {  
 using namespace std;  
 AVL<int> tree;  
  
 int queries;  
 string query;  
 cin >> queries;  
  
 for (int i = 0; i < queries; ++i) {  
 cin >> query;  
  
 try {  
 if (query == "insert") {  
 int element;  
 cin >> element;  
 tree.emplace(element);  
 } else if (query == "size") {  
 cout << tree.size() << endl;  
 } else if (query == "print") {  
 cout << tree << endl;  
 } else if (query == "contains") {  
 int element;  
 cin >> element;  
 cout << (tree.contains(element) ? "Yes" : "No") << endl;  
 } else if (query == "remove") {  
 int element;  
 cin >> element;  
 tree.remove(element);  
 }  
 }  
 catch (out\_of\_range&) {}  
 catch (exception&) {}  
 }  
 return 0;  
}

## [**Посилання на pull request**](https://github.com/artificial-intelligence-department/ai_programming_playground/pull/764) **4. Результати виконання завдань, тестування та фактично витрачений час:**

Завдання № VNS Lab 10

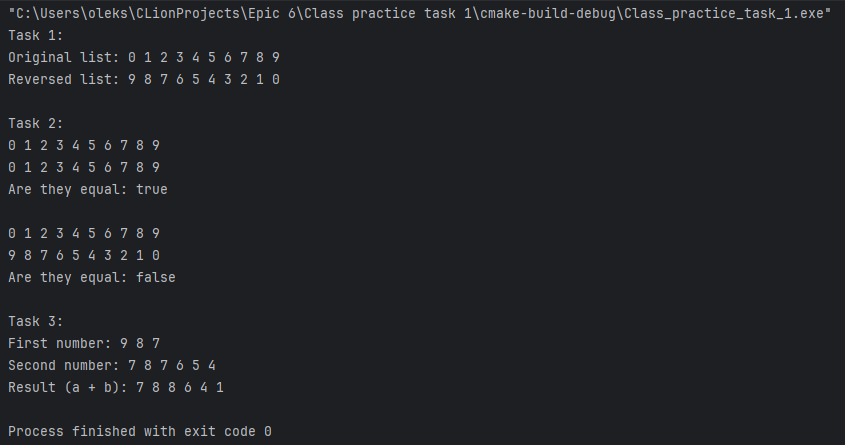
Час, витрачений на виконання завдання — 30 хвилин

Рисунок 9: Результати виконання VNS Lab 10

Рисунок 10: Результати виконання VNS Lab 10

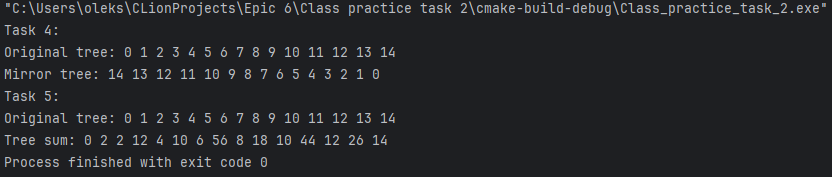
Завдання № 2.1 Class practice work (Linked lists)

Час, витрачений на виконання завдання — 40 хвилин

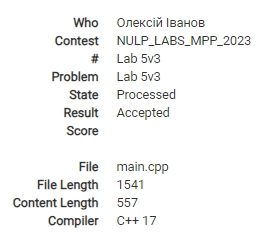
Рисунок 11: Результати виконання Class practice work (Linked lists)

Завдання № 2.2 Class practice work (Binary trees)

Час, витрачений на виконання завдання — 30 хвилин  
Завдання № 3 Algotester Lab 5

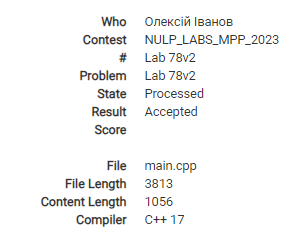
Рисунок 12: Результати виконання Class practice work (Binary trees)

Час, витрачений на виконання завдання — 40 хвилин

Рисунок 13: Результати виконання Algotester Lab 5

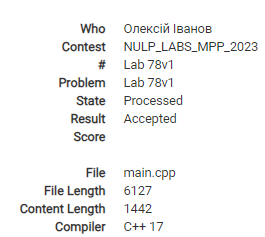
Завдання № 4 Algotester Lab 7-8

Час, витрачений на виконання завдання — 30 хвилин

Рисунок 14: Результати виконання Algotester Lab 7-8

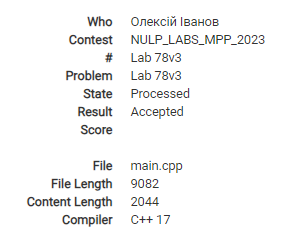
Завдання № 5 Self practice task 1

Час, витрачений на виконання завдання — 1 година

Рисунок 15: Результати виконання Self practice task 1

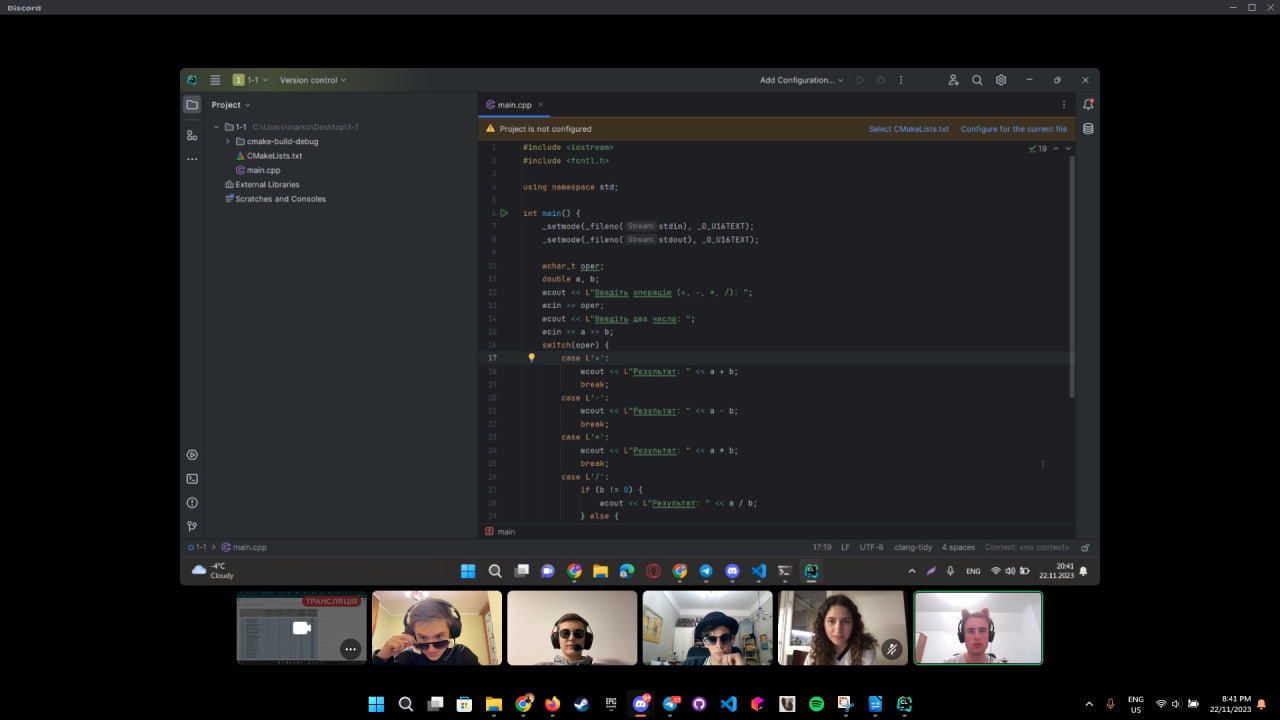
Завдання № 6 Self practice task 2

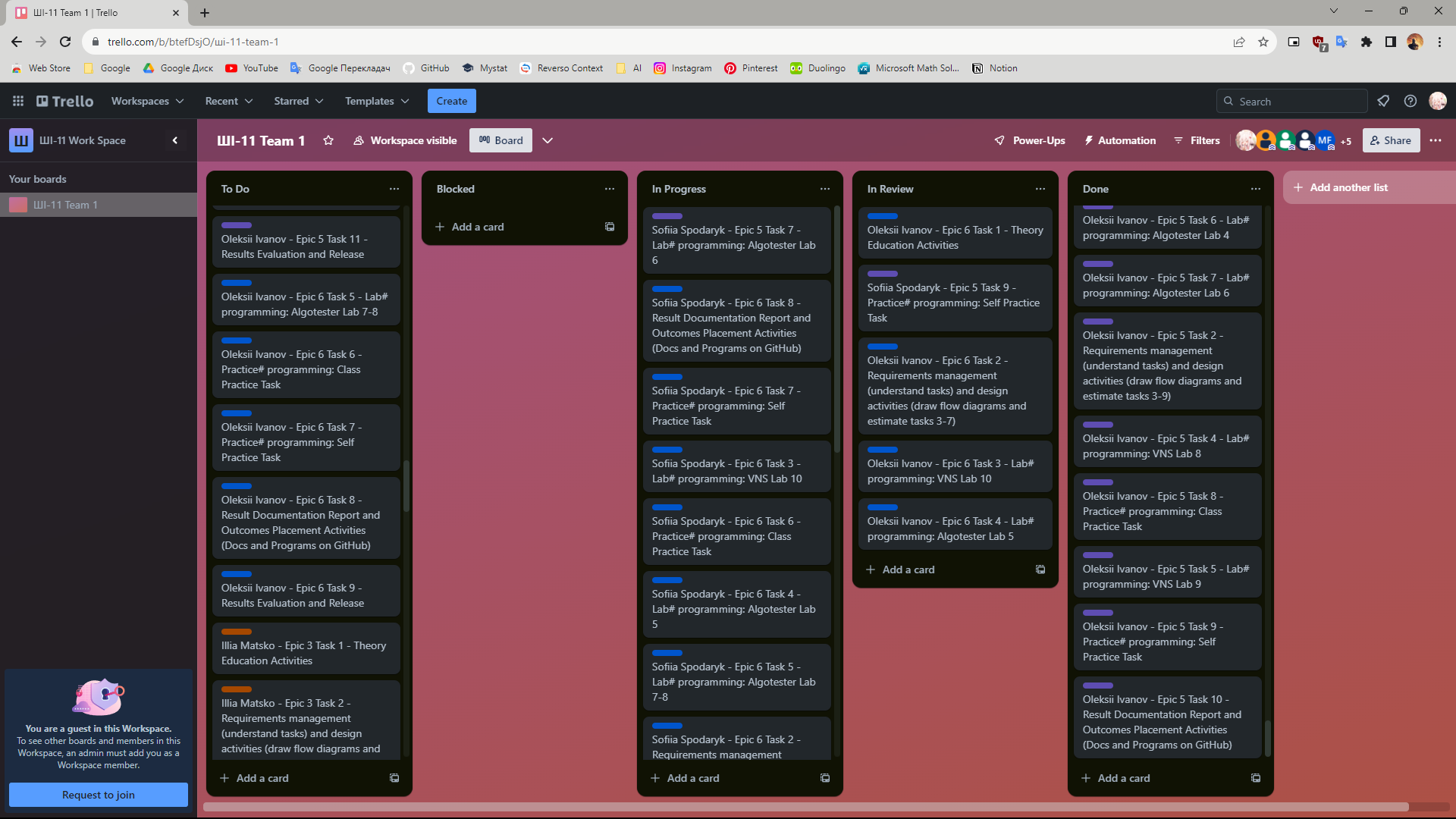
Час, витрачений на виконання завдання — 1 година

Рисунок 16: Результати виконання Self practice task 2

## **5. Кооперація з командою:**

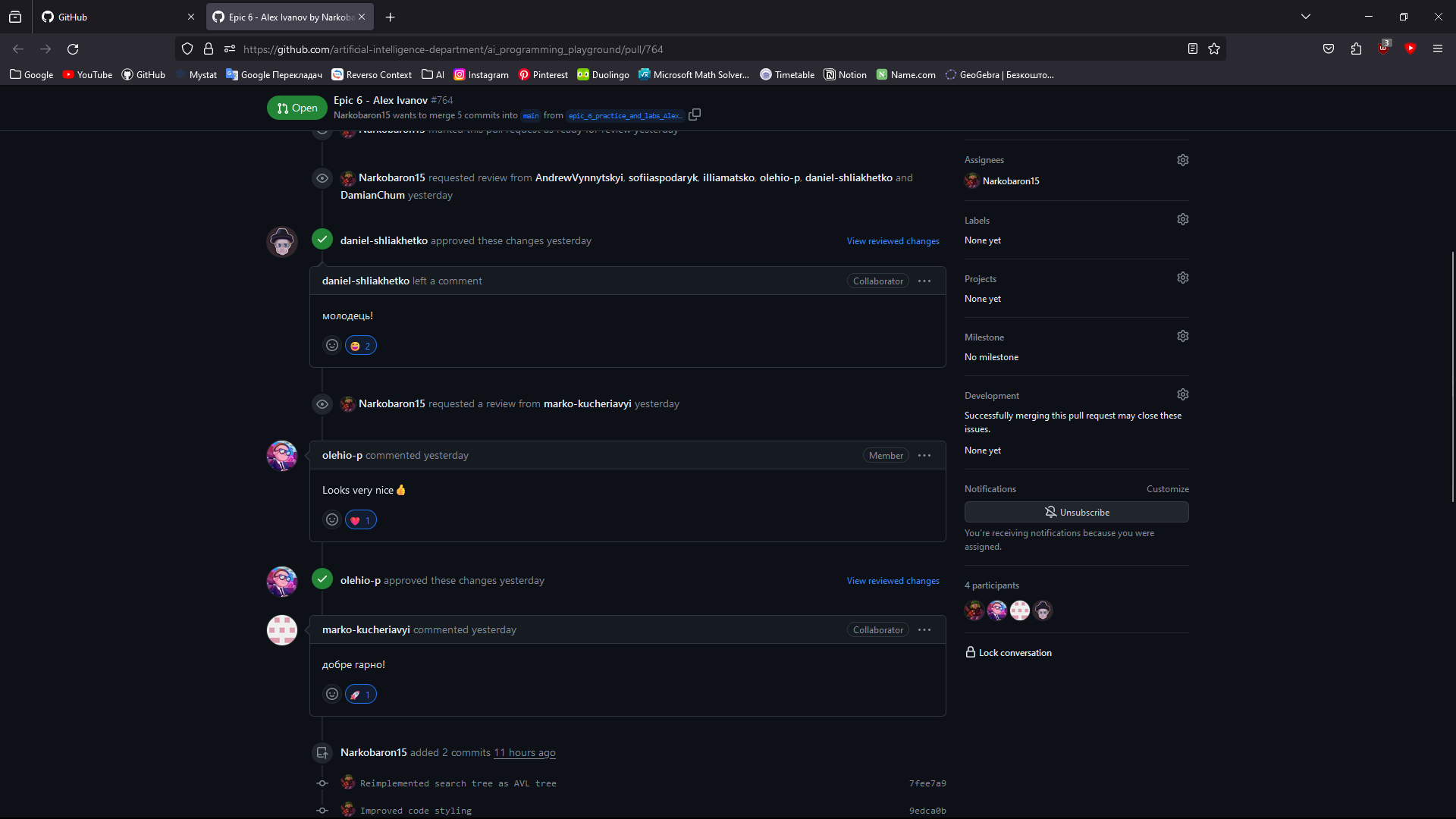
* Скрін зустрічі по обговоренню задач Епіку та Скрін прогресу по Трелло

Рисунок 17: Скріншот зустрічі команди

Рисунок 18: Скріншот прогресу в Trello

* Скрін з 2-му коментарями від учасників команди на пул реквесті з Ревю Роботи

# **Висновки:**

Рисунок 19: Скріншот коментарів до пул реквесту

Виконання лабораторної роботи дозволило ознайомитися з основними типами динамічних структур даних: стеком, чергою, списком, деревом. Були вивчені основні операції над динамічними структурами даних та закріплені навички реалізації алгоритмів на мові програмування С++. У процесі виконання лабораторної роботи були реалізовані динамічні структури даних - вектори за допомогою динамічних масивів, списки та дерева — власних вузлів. Дані колекції були згодом використанні для вирішення задач, поставлених у лабораторній роботі, що продемонструвало їх важливість та допомогло зрозуміти їх роботу зсередини.