

Звіт

Лабораторна робота №1

“Дослідження структур даних динамічний масив та зв’язний список”

Студента группи ДA-12

Краковича Павла Дмитровича

Київ – 2022

1. Мета роботи

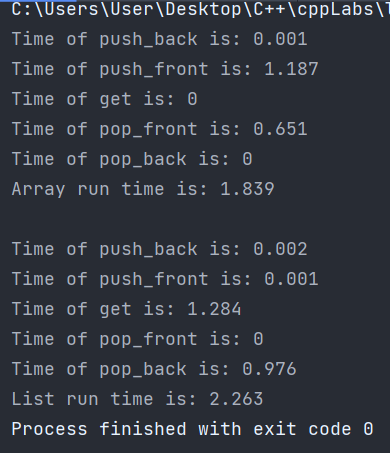
Ознайомитись і дослідити структури даних динамічний масив та зв’язний список. Поглибити розуміння роботи з динамічною пам’яттю та вказівниками. Набути навичок реалізації динамічного масиву та однозв’язного списку мовою програмування C++, порівняти час роботи основних операцій цих структур даних та дослідити їх асимптотичну складність.

2. Варіант роботи - №11

3. Хід виконання роботи

№1

* Структура Пара з матаналізу має наступні поля: ім’я викладача, номер пари, кількість присутніх студентів.
* Результат роботи програми:



* Лістинг(код) програми:
  + main.cpp

#include <iostream>  
#include <ctime>  
#include "LinkedList.h"  
#include "DynamicArray.h"  
  
using namespace std;  
  
  
int main() {  
 DynamicArray array;  
 LinkedList list;  
 srand(time(nullptr));  
 int i=0;  
 clock\_t startTimeVector = clock();  
 clock\_t startTimeVectorTest1 = clock();  
 while (i++ < 50000) {  
 matanClassArr data = {'t', 1, 1};  
 array.push\_back(&data);  
 }  
 clock\_t endTimeVectorTest1 = clock();  
 i=0;  
  
 clock\_t startTimeVectorTest2 = clock();  
 while (i++ < 10000) {  
 matanClassArr data = {'t', 1, 1};  
 array.push\_front(&data);  
 }  
 clock\_t endTimeVectorTest2 = clock();  
 i=0;  
  
 clock\_t startTimeVectorTest3 = clock();  
 while (i++ < 20000) {  
 int n = rand() % 59999 + 1;  
 array.get(n);  
 }  
 clock\_t endTimeVectorTest3 = clock();  
 i=0;  
  
 clock\_t startTimeVectorTest4 = clock();  
 while (i++ < 5000) {  
 array.pop\_front();  
 }  
 clock\_t endTimeVectorTest4 = clock();  
 i=0;  
  
 clock\_t startTimeVectorTest5 = clock();  
 while (i++ < 5000) {  
 array.pop\_back();  
 }  
 clock\_t endTimeVectorTest5 = clock();  
 clock\_t endTimeVector = clock();  
 i=0;  
  
 double secondsVectorTest1 = (double(endTimeVectorTest1 - startTimeVectorTest1)) / CLOCKS\_PER\_SEC;  
 cout << "Time of push\_back is: " << secondsVectorTest1;  
  
 double secondsVectorTest2 = (double(endTimeVectorTest2 - startTimeVectorTest2)) / CLOCKS\_PER\_SEC;  
 cout << "\nTime of push\_front is: " << secondsVectorTest2;  
  
 double secondsVectorTest3 = (double(endTimeVectorTest3 - startTimeVectorTest3)) / CLOCKS\_PER\_SEC;  
 cout << "\nTime of get is: " << secondsVectorTest3;  
  
 double secondsVectorTest4 = (double(endTimeVectorTest4 - startTimeVectorTest4)) / CLOCKS\_PER\_SEC;  
 cout << "\nTime of pop\_front is: " << secondsVectorTest4;  
  
 double secondsVectorTest5 = (double(endTimeVectorTest5 - startTimeVectorTest5)) / CLOCKS\_PER\_SEC;  
 cout << "\nTime of pop\_back is: " << secondsVectorTest5;  
  
 double secondsVector = (double(endTimeVector - startTimeVector)) / CLOCKS\_PER\_SEC;  
 cout << "\nArray run time is: " << secondsVector;  
  
  
 // list  
 clock\_t startTimeList = clock();  
 clock\_t startTimeListTest1 = clock();  
 while (i++ < 50000) {  
 matanClassList data = {'t', 1, 1};  
 list.push\_back(&data);  
 }  
 clock\_t endTimeListTest1 = clock();  
 i=0;  
  
 clock\_t startTimeListTest2 = clock();  
 while (i++ < 10000) {  
 matanClassList data = {'t', 1, 1};  
 list.push\_front(&data);  
 }  
 clock\_t endTimeListTest2 = clock();  
 i=0;  
  
 clock\_t startTimeListTest3 = clock();  
 while (i++ < 20000) {  
 int n = rand() % 59999 + 1;  
 list.get(n);  
 }  
 clock\_t endTimeListTest3 = clock();  
 i=0;  
  
 clock\_t startTimeListTest4 = clock();  
 while (i++ < 5000) {  
 list.pop\_front();  
 }  
 clock\_t endTimeListTest4 = clock();  
 i=0;  
  
 clock\_t startTimeListTest5 = clock();  
 while (i++ < 5000) {  
 list.pop\_back();  
 }  
 clock\_t endTimeListTest5 = clock();  
 clock\_t endTimeList = clock();  
  
  
 double secondsListTest1 = (double(endTimeListTest1 - startTimeListTest1)) / CLOCKS\_PER\_SEC;  
 cout << "\n\nTime of push\_back is: " << secondsListTest1;  
  
 double secondsListTest2 = (double(endTimeListTest2 - startTimeListTest2)) / CLOCKS\_PER\_SEC;  
 cout << "\nTime of push\_front is: " << secondsListTest2;  
  
 double secondsListTest3 = (double(endTimeListTest3 - startTimeListTest3)) / CLOCKS\_PER\_SEC;  
 cout << "\nTime of get is: " << secondsListTest3;  
  
 double secondsListTest4 = (double(endTimeListTest4 - startTimeListTest4)) / CLOCKS\_PER\_SEC;  
 cout << "\nTime of pop\_front is: " << secondsListTest4;  
  
 double secondsListTest5 = (double(endTimeListTest5 - startTimeListTest5)) / CLOCKS\_PER\_SEC;  
 cout << "\nTime of pop\_back is: " << secondsListTest5;  
  
 double secondsList = (double(endTimeList - startTimeList)) / CLOCKS\_PER\_SEC;  
 cout << "\nList run time is: " << secondsList;  
  
 return 0;  
}

* + DynamicArray.h

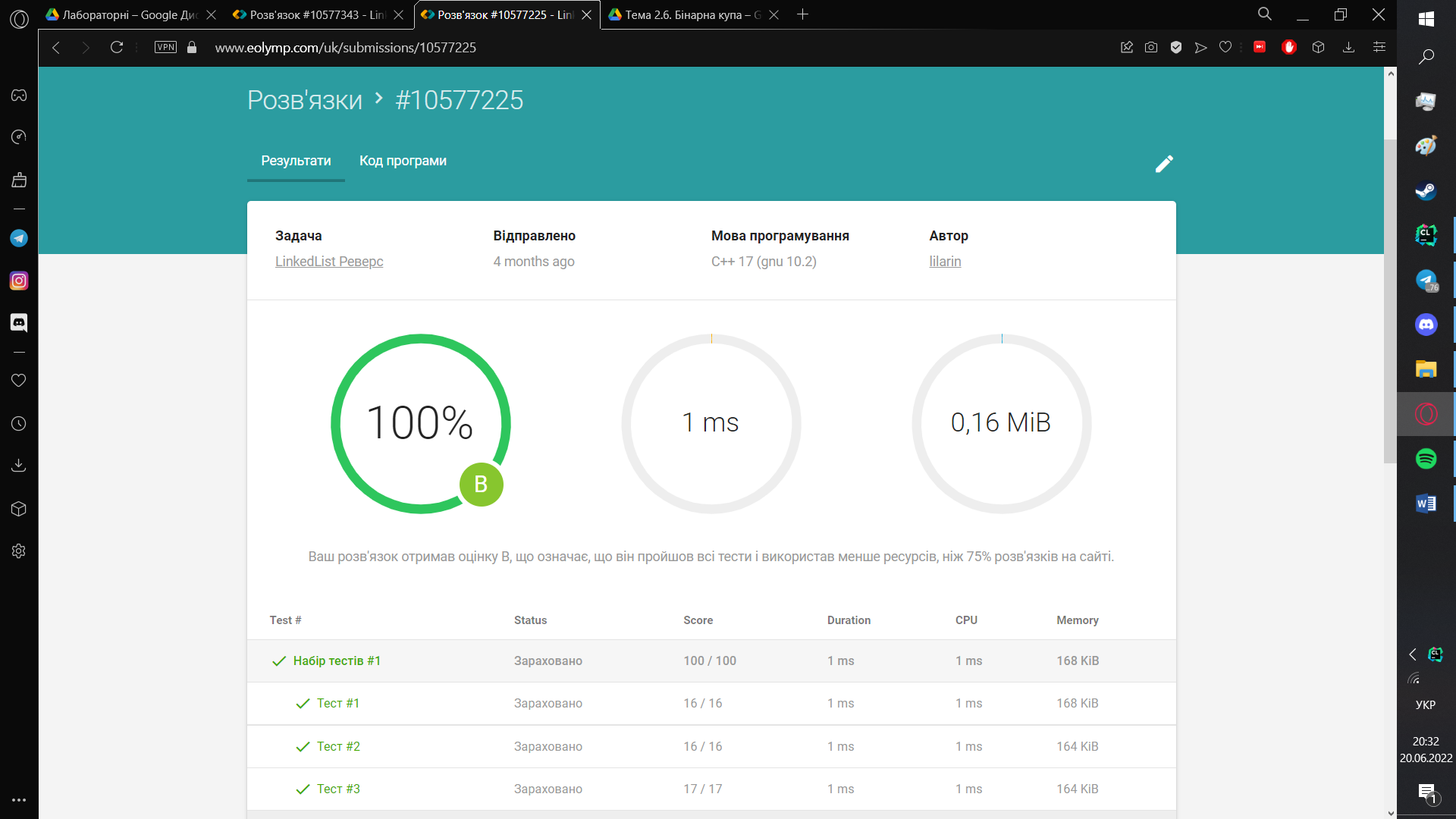
//  
// Created by User on 23.02.2022.  
//  
using namespace std;  
  
#ifndef MAIN\_CPP\_DYNAMICARRAY\_H  
#define MAIN\_CPP\_DYNAMICARRAY\_H  
  
struct matanClassArr {  
 char surnameOfLecturer;  
 int positionInOrder;  
 int amountOfStudents;  
};  
  
struct DynamicArray {  
 int max\_size = 1000,current\_size = 0;  
 matanClassArr\* arr = new matanClassArr[max\_size];  
 matanClassArr\* tempArr;  
 matanClassArr error;  
  
 void reallocateMemory() {  
 tempArr = new matanClassArr[max\_size];  
  
 for (int i = 0; i < current\_size; i++) {  
 tempArr[i] = arr[i];  
 }  
 delete [] arr;  
 max\_size = max\_size \* 2;  
 arr = new matanClassArr[max\_size];  
 for (int i = 0; i < max\_size / 2; i++) {  
 arr[i] = tempArr[i];  
 }  
 delete [] tempArr;  
 }  
  
 bool checkIsFull() const {  
 if (current\_size >= max\_size) {  
 return true;  
 }  
 return false;  
 }  
  
 bool checkIsEmpty() const {  
 if (current\_size == 0) {  
 return true;  
 }  
 return false;  
 }  
  
 void push\_back(matanClassArr \*data) {  
 if (checkIsFull()) {  
 reallocateMemory();  
 }  
 arr[current\_size] = \*data;  
 current\_size++;  
 }  
  
 void push\_front(matanClassArr \*data) {  
 if (checkIsFull()) {  
 reallocateMemory();  
 }  
 for (int i = current\_size; i > 0; i--) {  
 arr[i] = arr[i-1];  
 }  
 arr[0] = \*data;  
 current\_size++;  
 }  
  
 matanClassArr pop\_back() {  
 if (checkIsEmpty()) {  
 return error;  
 }  
 current\_size--;  
 matanClassArr temp = arr[current\_size];  
 return temp;  
 }  
  
 matanClassArr pop\_front() {  
 if (checkIsEmpty()) {  
 return error;  
 }  
 matanClassArr temp = arr[0];  
 for (int i = 0; i < current\_size-1; i++) {  
 arr[i] = arr[i+1];  
 if (i == current\_size-1) {  
 arr[current\_size] = temp;  
 }  
 }  
 current\_size--;  
 return temp;  
 }  
  
 matanClassArr get(int number) const {  
 matanClassArr temp = arr[number];  
 return temp;  
 }  
  
 void insert(matanClassArr \*data,int position) {  
 if (checkIsFull()) {  
 reallocateMemory();  
 }  
 for (int i = current\_size; i > position; i--) {  
 arr[i] = arr[i-1];  
 }  
 arr[position] = \*data;  
 current\_size++;  
 }  
  
 void remove(int position) {  
 for (int i = position; i < current\_size; i++) {  
 arr[i] = arr[i+1];  
 }  
 current\_size--;  
 }  
  
 int sizeOfArray() const {  
 return current\_size;  
 }  
  
 int capacity() const {  
 return max\_size;  
 }  
  
 void print() const {  
 matanClassArr temp;  
 for (int i = 0; i < current\_size; i++) {  
 temp = get(i);  
 cout << temp.surnameOfLecturer << " " << temp.positionInOrder << " " << temp.amountOfStudents << endl;  
 }  
 }  
  
 void clear() {  
 current\_size = 0;  
 delete [] arr;  
 }  
 ~DynamicArray() {  
 delete[] arr;  
 }  
};  
  
  
#endif //MAIN\_CPP\_DYNAMICARRAY\_H

* + LinkedList.h

//  
// Created by User on 23.02.2022.  
//  
  
#ifndef MAIN\_CPP\_LINKEDLIST\_H  
#define MAIN\_CPP\_LINKEDLIST\_H  
using namespace std;  
  
struct matanClassList {  
 char surnameOfLecturer;  
 int positionInOrder;  
 int amountOfStudents;  
};  
  
struct Node {  
 matanClassList data;  
 Node\* nextElement;  
 //Node\* previousElement;  
};  
  
struct LinkedList {  
 Node\* head;  
 Node\* tail;  
 int size = 0;  
  
 LinkedList() {  
 head = nullptr;  
 tail = nullptr;  
 }  
  
 void push\_front(matanClassList \*newData) {  
 Node\* node = new Node();  
 node->data = \*newData;  
 node->nextElement = head;  
 head = node;  
 size++;  
 if (size == 1) {  
 tail = head;  
 }  
 }  
  
 void push\_back(matanClassList \*newData) {  
 Node\* node = new Node();  
 node->data = \*newData;  
 if (tail != nullptr) {  
 tail->nextElement = node;  
 }  
  
 node->nextElement = nullptr;  
 tail = node;  
 size++;  
  
 if (size == 1) {  
 head = tail;  
 }  
 }  
  
 matanClassList pop\_front() {  
 matanClassList temp = head->data;  
 head = head->nextElement;  
 size--;  
 return temp;  
 }  
  
 matanClassList pop\_back() {  
 if (sizeOfList()==1) {  
 matanClassList temp = head->data;  
 return temp;  
 }  
 Node\* node = head;  
 matanClassList temp = tail->data;  
 while (node->nextElement->nextElement != nullptr) {  
 node = node->nextElement;  
 }  
 delete tail;  
 node->nextElement = nullptr;  
 tail = node;  
 size--;  
 return temp;  
 }  
  
 bool isEmpty() const {  
 if (size==0) {  
 return true;  
 }  
 return false;  
 }  
  
 void clear() {  
 Node\* node = head;  
 Node\* temp;  
 while (node != nullptr)  
 {  
 temp = node;  
 node = node->nextElement;  
 delete temp;  
 }  
 head = nullptr;  
 tail = nullptr;  
 size = 0;  
 }  
  
 void insert(matanClassList \*newData,int position) {  
 Node\* node = head;  
 Node\* newNode = new Node();  
 newNode->data = \*newData;  
  
 for (int i = 0; i < position - 1; i++) {  
 node = node->nextElement;  
 }  
 newNode->nextElement = node->nextElement;  
 node->nextElement = newNode;  
 if (node->nextElement == nullptr) {  
 tail = newNode;  
 }  
 size++;  
 }  
  
 matanClassList remove(int position) {  
 Node\* node = head;  
 Node\* temp;  
 for (int i = 0; i < position - 1; i++) {  
 node = node->nextElement;  
 }  
 temp = node;  
 node = node->nextElement;  
 temp->nextElement = node->nextElement;  
 matanClassList returnObject = node->data;  
 delete node;  
  
 size--;  
 return returnObject;  
 }  
  
 void print() const {  
 Node\* current = head;  
 while (current != nullptr) {  
 cout << current->data.surnameOfLecturer << " " << current->data.positionInOrder << " " << current->data.amountOfStudents << endl;  
 current = current->nextElement;  
 }  
 }  
  
 matanClassList get(int position) const {  
 Node\* node = head;  
 for (int i = 0; i <= position - 1; i++) {  
 node = node->nextElement;  
 }  
 matanClassList temp = node->data;  
 return temp;  
 }  
  
 int sizeOfList() const {  
 return size;  
 }  
  
 ~LinkedList() {  
 clear();  
 }  
};  
  
#endif //MAIN\_CPP\_LINKEDLIST\_H

№2(Додаткове завдання)

* Розв’язати задачу “Розворот однозв’язного списку”:
* Результат роботи програми:



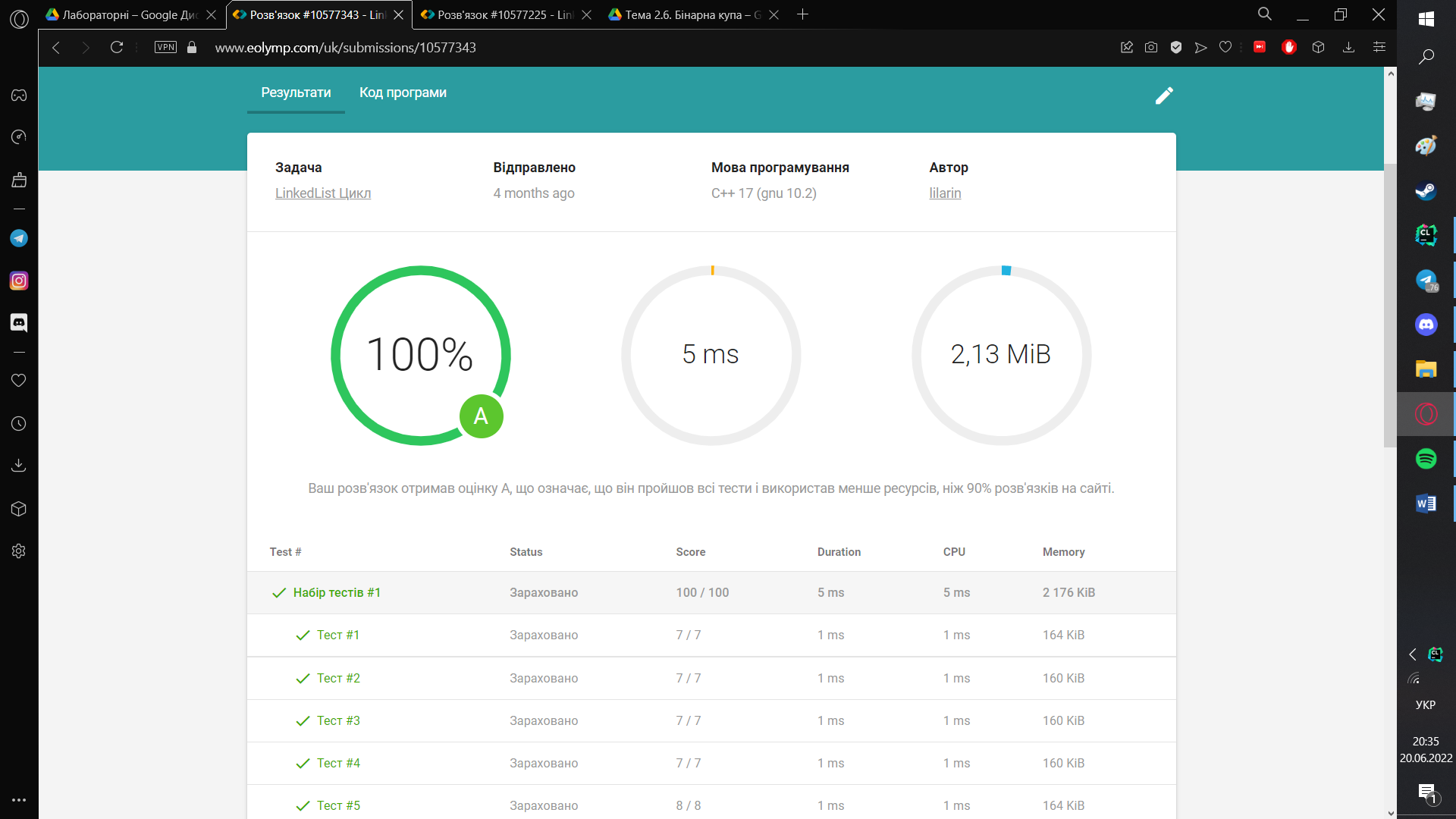
(робив програму 4 місяці тому, а протокол тільки зараз, тому на скріні програма зроблена так давно)

* Лістинг(код) програми:

ListNode\* reverse(ListNode\* head) {  
 if (head == nullptr || head->next == nullptr) {  
 return head;  
 }  
 ListNode\* node = head;  
 ListNode\* nextNode = head->next;  
 head->next = nullptr;  
  
 while (nextNode != nullptr) {  
 ListNode\* temp = nextNode->next;  
 nextNode->next = node;  
 node = nextNode;  
 nextNode = temp;  
 }  
 return node;  
}

№3(Додаткове завдання)

* Розв’язати задачу “Перевірка однозв’язного списку на циклічність”:
* Результат роботи програми:



* Лістинг(код програми)

int hasCycle(ListNode \*head) {  
 ListNode\* turtle = head;  
 ListNode\* rabbit = head;  
 while (rabbit != nullptr && rabbit->next != nullptr) {  
 turtle = turtle->next;  
 rabbit = rabbit->next->next;  
 if (turtle == rabbit) {  
 return 1;  
 }  
 }  
 return 0;  
}

Висновки

В процесі виконання лабораторної роботи №1 я ознайомився, до-слідив та реалізував структури «даних динамічний масив» та «зв’язний» список. Навчився правильно виділяти пам’ять для структури даних «динамічний масив». В ході порівняння швидкості роботи я з’ясував, що моя реалізація динамічного масиву працює трішки швидше, ніж зв’язний список. Хоча, їхні методи окремо мають кардинально різний час виконання через особливості побудови та логіки.

Виконуючи додаткові завдання у системі eolymp, я покращив свої навички ти вирішив дані завдання. Також, я дослідив пам’ять на стеку та купі та визначив навчився визначати асимптотичну складність методів функцій програми.

Кожна структура даних має сильні та слабкі сторони, які впливають на продуктивність залежно від завдання. Масиви дозволяють випадковий доступ і вимагають менше пам’яті на елемент (не потребують місця для покажчиків), водночас не мають ефективності для операцій вставки/видалення та виділення пам’яті. Навпаки, зв’язані списки є динамічними та мають швидший час вставки/видалення. Однак зв’язаний список має більш повільний час пошуку, а вказівники вимагають додаткової пам’яті на елемент у списку.

Контрольні запитання

1) Назвіть основні переваги та недоліки динамічного масиву та однозв’язного списку, для яких задач їх краще використовувати?

*Переваги динамічних масивів:*

*До переваги динамічних масивів можна віднести час пошуку. Масиви зберігають елементи в безперервних місцях пам'яті. Це означає, що будь-який елемент можна отримати, додавши зміщення до базового значення масиву або розташування першого елемента.* *Оскільки немає різниці в часі між пошуком другого або останнього елемента в масиві, масиви мають постійний час пошуку або Великий O з одиниці (O(1)), що дуже швидко.*

*Недоліки динамічних масивів:*

*До недоліків динамічних масивів можна віднести кількість витраченої пам’яті через те, що вона може бути втрачена даремно, якщо задана на початку роботи програма кількість пам'яті не використана.*

*Також, недоліком є повільна вставка та видалення.Час вставки для масивів Великий O з n (O(n)), оскільки n елементів необхідно зрушити. Однак асимптотика вставки та видалення з кінця масиву є O(1).*

*Переваги зв’язних списків:*

*З точки зору розподілу пам’яті, зв’язані списки є більш ефективними, ніж масиви. На відміну від масивів, розмір зв’язаного списку не визначено заздалегідь, що дозволяє зв’язаному списку збільшуватися або зменшуватися під час виконання програми. Це можливо, тому що для вставки або видалення зі зв’язаного списку вказівники потрібно відповідно оновити. Як правило, для вставки вузла у зв’язаний список потрібно оновити покажчики після ініціалізації нового вузла.*

*Також, перевагою списку в час вставки та видалення, бо вставка нового вузла на початок або кінець зв’язаного списку займає постійний час (O(1)), оскільки єдиними кроками є ініціалізація нового вузла, а потім оновлення покажчиків. Також, якщо у реалізації існує вставка хвостового вказівника (подібний до головного вказівника) у кінець зв’язаного списку також O(1). Однак вставка в середину зв’язаного списку займає лінійний час (O(n)), оскільки потрібна ітерація над n елементами, щоб дістатися до правильного місця перед вставкою вузла. Аналогічно, видалення вузлів на початку і в кінці зв’язаного списку займає постійний час, тоді як видалення вузла в середині зв’язаного списку займає лінійний час.*

*Недоліки зв’язних списків:*

*Зв’язні списки мають менший час пошуку, ніж масиви, оскільки випадковий доступ не дозволяється. На відміну від масивів, де елементи можна шукати за індексом, зв’язаний список потребує ітерації. Це означає, що якщо ви хочете отримати дані на десятому вузлі, вказівник голови можна використовувати, щоб дістатися до першого вузла, вказівник на першому вузлі можна використовувати, щоб дістатися до другого вузла, і так далі до десятого вузол досягнуто. Це означає, що чим більше вузлів необхідно повторити під час пошуку вузла, тим довший час пошуку. Це означає, що час пошуку для зв’язаного списку є лінійним часом або великим O з n (O(n)).*

2) Чому операція додавання елементу в кінець динамічного масиву може займати різний час?

*Не дивлячись на те, що для додавання нового елементу майже завжди потрібно лише знати поточний розмір та додати елемент в кінець масиву, однак, можливий варіант, за якого для додавання нового елементу потрібно буде зробити перевиділення пам’яті для масиву, а тільки після цього додати новий елемент.*

3) Навіщо використовувати вказівник на хвіст в реалізації однозв’язного списку?

*Це допомагає зробити асимптотику видалення останнього елементу O(1). Інакше, асимптотика буде O(n), бо потрібно буде пройти по всім елементам списку.*

4) Чим відрізняється пам’ять на стеку та на купі? Що таке сміття у пам’яті та як цього не допустити?

*Стек – це дуже швидке сховище пам'яті, що працює за принципом LIFO та кероване процесором. Але ці переваги призводять до обмеженого розміру стека та спеціального способу отримання значень. Щоб уникнути цих обмежень, можна користуватися купою - вона дозволяє створювати динамічні та глобальні змінні - але керувати пам'яттю повинен або збирач сміття, або сам програміст.*

*Купа – сховище пам'яті, розташоване у ОЗУ. Воно допускає динамічне виділення пам'яті та працює не так, як стек. По суті йдеться про простий склад для ваших змінних. Коли ви виділяєте тут ділянку пам'яті для зберігання, до неї можна звертатися як у потоці, так і у всьому додатку загалом (саме так і визначаються змінні глобального типу). Після завершення роботи програми всі виділені ділянки звільняються.*

*Розмір купи задається під час запуску програми, однак, на відміну від того, як працює стек, у купі розмір обмежений лише фізично, що дозволяє створювати змінні динамічного типу.*

*Якщо порівнювати, знову ж таки, про те, як працює стек, то купа функціонує повільніше, тому що змінні розкидані по пам'яті, а не знаходяться вгорі стека. Проте цей факт не зменшує значення купи, і якщо вам треба працювати з глобальними або динамічними змінними, вона більше підходить. Однак керувати пам'яттю тоді повинен програміст або збирач сміття.*

*Коли створені об’єкти класу здійснюють роздільну оперативну пам’ять new. Після того, як об’єкт виконує свою роботу, він (можливо) вже не використовується в програмі (але пам’ять все ще займає). Якщо у програмі багато об’єктів, то може виникнути окремі коливання пам’яті не вистачати для створення наступного об’єкту.*

5) Як можна використати код зв’язного списку для реалізації стека та черги? У чому полягає різниця між стеком / чергою, реалізованих на статичних масивах та на зв’язних списках?

*Щоб використати код зв’язного списку для стеку та черги, потрібно додати логіку роботи додавання елементу та його видалення зі списку. Різниця між таким стеком та чергою буде полягати в методі видалення.  
Стеки / черги на масивах та на зв’язних списках будуть відрізнятися методами роботи з ними та асимптотикою методів, бо зв’язний список та масив мають свої переваги та недоліки.*

6) Для чого потрібні двозв’язні списки, в чому їх переваги та недоліки в порівнянні з однозв’язними?

*Двозв’язні списки, навідміну від однозв’язних дозволяють користувачу ітеруватися в двох напрямках (вперед та назад), що покращує асимптотику методів (вставка та видалення), але потребує зберігати вдічі більше вказівників, щоб збільшує кількість необхідної пам’яті.*