МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЖИТОМИРСЬКИЙ ДЕРЖАВНИЙ ТЕХНОЛОГІЧНИЙ УНІВЕРСИТЕТ

ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП’ЮТЕРНИХ ТЕХНОЛОГІЙ

Кафедра інженерії програмного забезпечення

**Звіт з лабораторних робіт**

**з предмету**

**«Алгоритми та структури даних»**

студентки І курсу групи ПІ-60

спеціальності 121 «Інженерія програмного забезпечення»

Харипончук Марії Миколаївни

Житомир – 2019

**ЗМІСТ**

[ЛАБОРАТОРНА РОБОТА № 1 3](#_Toc9265326)

[ЛАБОРАТОРНА РОБОТА № 2 9](#_Toc9265327)

[ЛАБОРАТОРНА РОБОТА № 3 11](#_Toc9265328)

[ЛАБОРАТОРНА РОБОТА № 4 15](#_Toc9265329)

[ЛАБОРАТОРНА РОБОТА № 5 18](#_Toc9265330)

[ЛАБОРАТОРНА РОБОТА № 6 21](#_Toc9265331)

ЛАБОРАТОРНА РОБОТА № 1

**Работа с базовыми типами данных**

***Цель работы:*** получить практические навыки по работе с базовыми типами

данных (простыми и составными типами данных).

**Порядок выполнения работы:**

**Завдание 1**: Записать и заполнить структуру данных хранения текущего времени (включая секунды) и даты в наиболее компактном виде, используя перечисления, объединения и др. типы данных. Определить объем памяти, занимаемой структурой данных. Вывести дату в текстовом виде.

Сначала была создана структура Date, которая хранит данные о текущем времени и дате. На каждое поле структуры было выделено определенное количество бит.

Например, для переменной nHours (час) выделяем 5 битов, так как ее значение может находиться в диапазоне 0..24, а 5 бит позволяют сохранять значение 0 .. (25 - 1), то есть 0..31. По той же логике определяем разрядность для других полей:

struct Date

{

unsigned char nHours : 5;

unsigned char nMinutes : 6;

unsigned char nSeconds : 6;

unsigned char nMonthDay : 5;

unsigned char nMonth : 5;

unsigned char nYear : 8;

}date;

Значения переменных для структур инициализируем в главной функции main(), каждому полю структуры присвоено значение, которое храниться в компьютере, с помощью функции time():

#include <time.h>

int main()

{ time\_t Time = time(NULL);

tm \*aTime = localtime(&Time);

date.nHours = aTime->tm\_hour;

date.nMinutes = aTime->tm\_min;

date.nSeconds = aTime->tm\_sec;

date.nMonthDay = aTime->tm\_mday;

date.nMonth = aTime->tm\_mon;

}

И выводим дату и время на консоль:

printf("Дата и время:\t%.2d/%.2d/2019 %.2d:%.2d:%.2d", date.nMonthDay, date.nMonth + 1, date.nHours, date.nMinutes, date.nSeconds);

А также определено и выведено объем памяти, занимаемой структурой данных.

printf("\nРазмер: %d байт - %d байт", sizeof(Date), sizeof(tm));

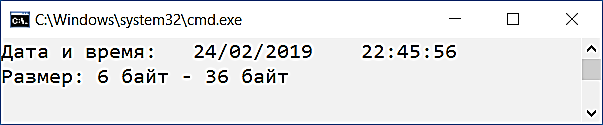


Рисунок 1.1- Вывод даты и времени на экран консоли

**Завдание 2:** Реализовать ввод целочисленного значения типа signed short. Определить знак и значение, используя: 1) определение структуры данных; 2) логические операции; 3) арифметические операции.

Сначала было создано объединение Num, которое содержит переменную целого числа и структуру битового вида этого числа(для уменьшения объема памяти, который занимает структура). На каждое поле структуры было выделено 1 бит, ибо значение signed short находится в диапазоне 0..255, то есть 0..215. Для sign нужен только 1 бит, ведь в нем будет находиться значение 1 при положительном числе и 0 при отрицательном.

union Num {

struct {

unsigned char n1 : 1;

unsigned char n2 : 1;

unsigned char n3 : 1;

unsigned char n4 : 1;

unsigned char n5 : 1;

unsigned char n6 : 1;

unsigned char n7 : 1;

unsigned char n8 : 1;

unsigned char n9 : 1;

unsigned char n10 : 1;

unsigned char n11 : 1;

unsigned char n12 : 1;

unsigned char n13 : 1;

unsigned char n14 : 1;

unsigned char n15 : 1;

unsigned char sign : 1;

}bit;

signed short wholeNum;

}num;

Программа считывает целое число и выводит его значение и знак. Чтобы определить знак и значение с помощью логических операций, используем операцию if. В ней проверяем или введеное число равно нулю: если да, то как знак выводим "+", в противном случае "-".

int main()

{

printf("Введите число: ");

scanf\_s("%d", &num.wholeNum);

printf("\n Значение: %d", num.wholeNum);

printf("\n Битовый код: %d%d%d%d %d%d%d%d %d%d%d%d %d%d%d%d", num.bit.sign, num.bit.n15, num.bit.n14, num.bit.n13, num.bit.n12, num.bit.n11, num.bit.n10, num.bit.n9, num.bit.n8, num.bit.n7, num.bit.n6, num.bit.n5, num.bit.n4, num.bit.n3, num.bit.n2, num.bit.n1);

printf("\n Знак: (%d) ", num.bit.sign);

if(num.bit.sign == 0)

printf("\"+\"");

else

printf("\"-\"");

}

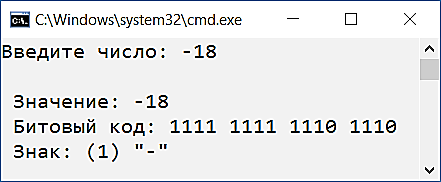


Рисунок 1.2.1 - Определение знака и значения числа

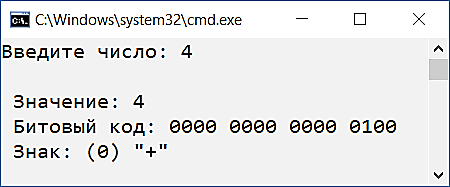


Рисунок 1.2.2 - Определение знака и значения числа

**Завдання 3**:

Выполнить операции:

а) 5+127; б) 2-3; в) -120-34; г) (unsigned char)(-5); д) 56 & 38; е) 56 | 38.

Операции выполнять для переменных типа signed char. Выполнить в ручную проверку результата. Пояснить результат, используя двоичное представления численного результата.

Сначала было создано объединение Num, которое содержит переменную целого числа и структуру битового вида этого числа. На каждое поле структуры было выделено 1 бит (как и во втором задании):

union Num {

struct {

unsigned char n1 : 1;

unsigned char n2 : 1;

unsigned char n3 : 1;

unsigned char n4 : 1;

unsigned char n5 : 1;

unsigned char n6 : 1;

unsigned char n7 : 1;

unsigned char n8 : 1;

unsigned char n9 : 1;

unsigned char n10 : 1;

unsigned char n11 : 1;

unsigned char n12 : 1;

unsigned char n13 : 1;

unsigned char n14 : 1;

unsigned char n15 : 1;

unsigned char sign : 1;

}bit;

short wholeNum;

}num;

Создано функцию побитового вывода числа:

void printInBites(int number) {

num.wholeNum = number;

printf("\n %d", num.wholeNum);

printf("\t\t %d%d%d%d %d%d%d%d %d%d%d%d %d%d%d%d", num.bit.sign, num.bit.n15, num.bit.n14, num.bit.n13, num.bit.n12, num.bit.n11, num.bit.n10, num.bit.n9, num.bit.n8, num.bit.n7, num.bit.n6, num.bit.n5, num.bit.n4, num.bit.n3, num.bit.n2, num.bit.n1);

}

Выведено в главной функции результаты обчислений примеров:

int main()

{

printf(" Значение: \t Битовый код:");

printInBites((signed char)(5 + 127));

printInBites((signed char)(2 - 3));

printInBites((signed char)(-120 - 34));

printInBites((unsigned char)((signed char)(-5)));

printInBites((signed char)(56 & 38));

printInBites((signed char)(56 | 38));

}

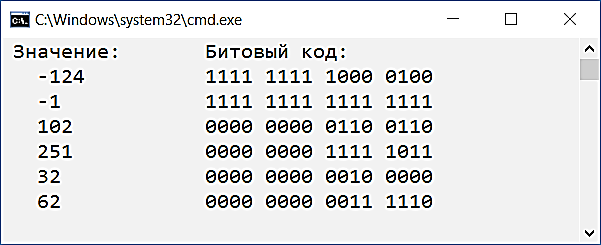


Рисунок 1.3 - Результат роботы консольной программы

**Завдание 4**: Записать и заполнить структуру данных хранения действительного числа типа float в наиболее компактном виде, используя перечисления, объединения и др. типы данных. Реализовать вывод на дисплей: 1) значения побитово; 2) значения побайтово; 3) знака, мантиссы и порядка значения. Выполнить в ручную проверку результата. Определить объем памяти, занимаемой структурой.

Сначала было создано объединение Num, которое содержит переменную дробового числа и вложенные структуры в структуру битового вида этого числа. На каждое поле структуры было выделено 1 бит:

union num {

struct {

struct {

unsigned char n1 : 1;

unsigned char n2 : 1;

unsigned char n3 : 1;

unsigned char n4 : 1;

unsigned char n5 : 1;

unsigned char n6 : 1;

unsigned char n7 : 1;

unsigned char n8 : 1;

}b1;

struct {

unsigned char n9 : 1;

unsigned char n10 : 1;

unsigned char n11 : 1;

unsigned char n12 : 1;

unsigned char n13 : 1;

unsigned char n14 : 1;

unsigned char n15 : 1;

unsigned char n16 : 1;

}b2;

struct {

unsigned char n17 : 1;

unsigned char n18 : 1;

unsigned char n19 : 1;

unsigned char n20 : 1;

unsigned char n21 : 1;

unsigned char n22 : 1;

unsigned char n23 : 1;

unsigned char n24 : 1;

}b3;

struct {

unsigned char n25 : 1;

unsigned char n26 : 1;

unsigned char n27 : 1;

unsigned char n28 : 1;

unsigned char n29 : 1;

unsigned char n30 : 1;

unsigned char n31 : 1;

unsigned char n32 : 1;

}b4;

} bit;

float wholeNum;

}Num;

В главной функции выведено побитовый код этого числа и размер каждого бита, мантиссу, знак и размер всей структуры:

int main()

{

Num.wholeNum = -37501;

printf(" %d%d%d%d%d%d%d%d", Num.bit.b4.n32, Num.bit.b4.n31, Num.bit.b4.n30, Num.bit.b4.n29, Num.bit.b4.n28, Num.bit.b4.n27, Num.bit.b4.n26, Num.bit.b4.n25);

printf("\t%d%d%d%d%d%d%d%d", Num.bit.b3.n24, Num.bit.b3.n23, Num.bit.b3.n22, Num.bit.b3.n21, Num.bit.b3.n20, Num.bit.b3.n19, Num.bit.b3.n18, Num.bit.b3.n17);

printf("\t%d%d%d%d%d%d%d%d", Num.bit.b2.n16, Num.bit.b2.n15, Num.bit.b2.n14, Num.bit.b2.n13, Num.bit.b2.n12, Num.bit.b2.n11, Num.bit.b2.n10, Num.bit.b2.n9);

printf("\t%d%d%d%d%d%d%d%d\n", Num.bit.b1.n8, Num.bit.b1.n7, Num.bit.b1.n6, Num.bit.b1.n5, Num.bit.b1.n4, Num.bit.b1.n3, Num.bit.b1.n2, Num.bit.b1.n1);

printf(" 1байт - %d \t2байт - %d \t3байт - %d \t4байт - %d\n", Num.bit.b4, Num.bit.b3, Num.bit.b2, Num.bit.b1);

printf(" Мантисса - %d%d%d%d%d%d%d", Num.bit.b3.n23, Num.bit.b3.n22, Num.bit.b3.n21, Num.bit.b3.n20, Num.bit.b3.n19, Num.bit.b3.n18, Num.bit.b3.n17);

printf(" %d%d%d%d%d%d%d%d", Num.bit.b2.n16, Num.bit.b2.n15, Num.bit.b2.n14, Num.bit.b2.n13, Num.bit.b2.n12, Num.bit.b2.n11, Num.bit.b2.n10, Num.bit.b2.n9);

printf(" %d%d%d%d%d%d%d%d\n", Num.bit.b1.n8, Num.bit.b1.n7, Num.bit.b1.n6, Num.bit.b1.n5, Num.bit.b1.n4, Num.bit.b1.n3, Num.bit.b1.n2, Num.bit.b1.n1);

printf(" Знак - %d \n Размер - %dбайт", Num.bit.b4.n32, sizeof(num));

}

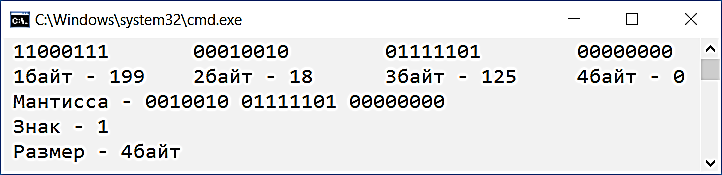


Рисунок 1.4 - Результат роботы консольной программы

***Выводи:*** в результате выполнения лабораторной работы получено практические навыки по работе с базовыми типами данных (простыми и составными типами данных).

ЛАБОРАТОРНА РОБОТА № 2

**Генерування послідовності псевдовипадкових значень**

***Мета:*** ознайомитись з методами генерування випадкових чисел, а також

формуванням та обробкою масивів даних.

**Хід роботи:**

**Завдання 1**: Розробити програму \* генерування цілочислової послідовності псевдовипадкових значень (за допомогою конгруентного методу\*) та виконати обробку отриманого масиву даних наступним чином:

- розрахувати частоту інтервалів появи випадкових величин (інтервал дорівнює 1);

- розрахувати статистичну імовірність появи випадкових величин;

- розрахувати математичне сподівання випадкових величин;

- розрахувати дисперсію випадкових величин;

- розрахувати середньоквадратичне відхилення випадкових величин.

**Хід виконання завдань:**

1. Спочатку, пишемо функцію, яка буде генерувати псевдовипадкове число лінійним конгруентним методом. Вона приймає мінімальне та максимальне значення діапазону згенерованих чисел та повертає саме число.

int GetRand(int min, int max)

{

static int X = 1;

long long M = pow(2, 32);

X = abs((A \* X + C) % M);

return min + X % (max - min + 1);

}

2. Далі виводимо всі числа на екран.

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

{

arr[i] = GetRand(MIN, MAX);

printf("%d ", arr[i]);

}

3. Та рахуємо кількість повторень (частоту інтервалів) кожного числа з діапазону.

int frequency[SIZE];

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

frequency[i] = 0;

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

frequency[arr[i]]++;

printf("\nЧастота інтервалів: \n");

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

printf(" %d => %d\n", i, frequency[i]);

}

4. Розраховуємо систематичну ймовірність сподівання випадкових величин.

float staticProbability[SIZE];

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

staticProbability[i] = 0;

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

staticProbability[i] = (float)frequency[i] / N;

printf("\nСтатистична імовірність: \n");

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

printf(" %d => %.3f\n", i, staticProbability[i]);

}

5. Знаходимо математичне сподівання – це середнє значення чисел з вибірки.

float mathExpectation = 0;

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

mathExpectation += i \* frequency[i];

printf("\nМатематичне сподівання: \n");

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

printf(" %d => %.4f\n", i, staticProbability[i]);

}

printf("\nМатиматичне сподівання: %.3f\n", mathExpectation);

6. Визначаємо дисперсію відхилення величин, чим більше значення, тим більше діапазон генерування чисел відносно середнього значення вибірки.

float varianceOfRandNumbers = 0;

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

varianceOfRandNumbers += pow(i - mathExpectation, 2) \* frequency[i];

printf("\nДисперсія випадкової величини: %.3f\n", varianceOfRandNumbers);

7. Далі розраховуємо середньоквадратичне відхилення випадкових величин, що показує, на скільки в середньому відхиляються конкретні значення вибірки від середнього їх значення.

float meanSquareDeviation = sqrt(mathExpectation);

printf("\nСередньоквадратичне відхилення: %.3f\n", meanSquareDeviation);

8. В кінці пишемо меню, де зможемо вибрати, яку з функцій хочемо виконати.

static int DisplayMenu()

{

int menu;

printf("\n-------------Головне меню-------------\n");

printf(" Виберiть одну iз запропонованих опцiй:\n");

printf(" 1 - Згенерувати та вивести масив цілочислової послідовності псевдовипадкових значень\n")

printf(" 2 - Розрахувати частоту інтервалів появи випадкових величин\n");

printf(" 3 - Розрахувати статистичну імовірність появи випадкових величин\n");

printf(" 4 - Розрахувати математичне сподівання випадкових величин\n");

printf(" 5 - Розрахувати дисперсію випадкових величин\n");

printf(" 6 - Розрахувати середньоквадратичне відхилення випадкових величин\n");

printf(" 0 - Вихiд\n");

printf("-> ");

scanf\_s("%d", &menu);

return menu;

}

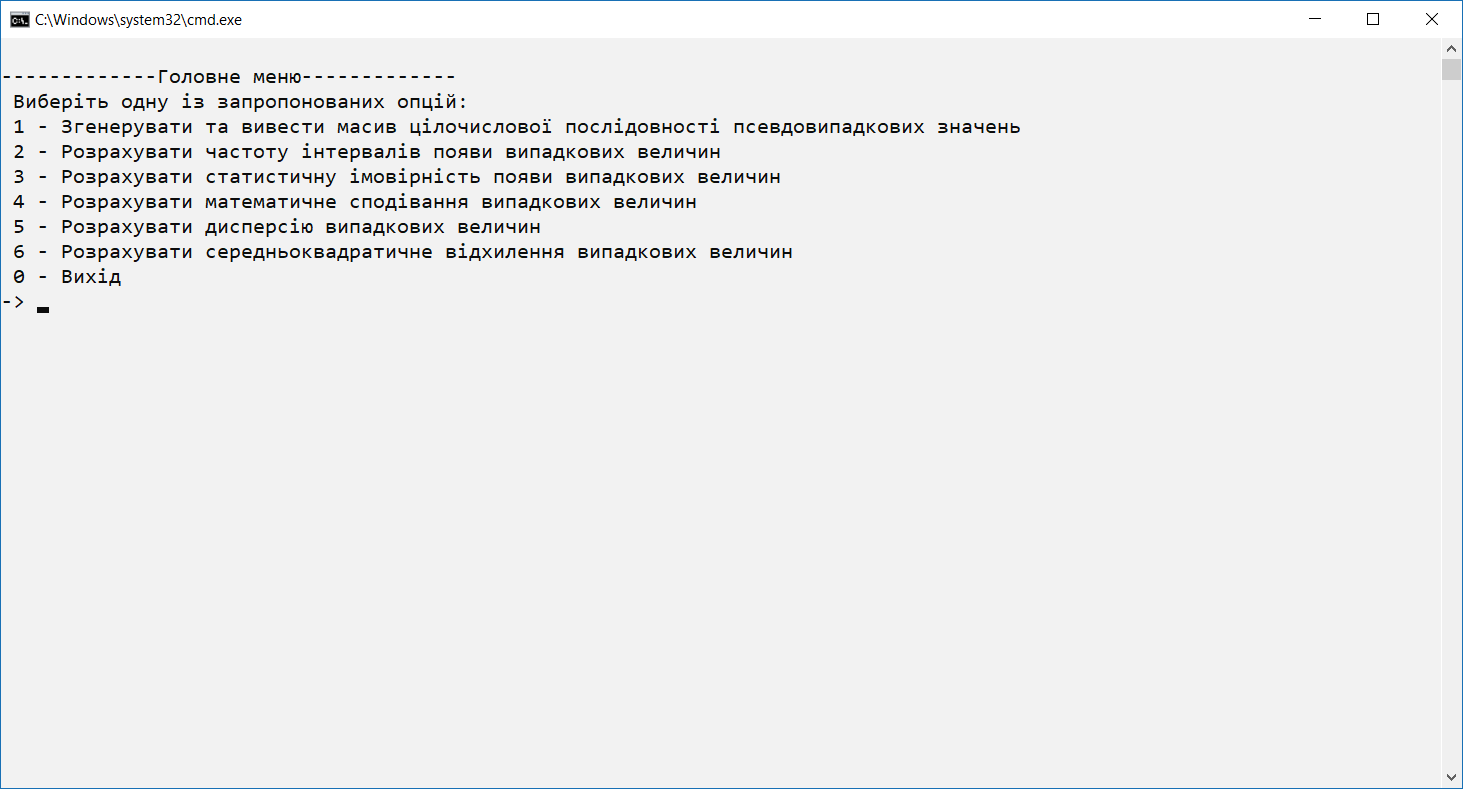


Рисунок 2.1 - Головне меню

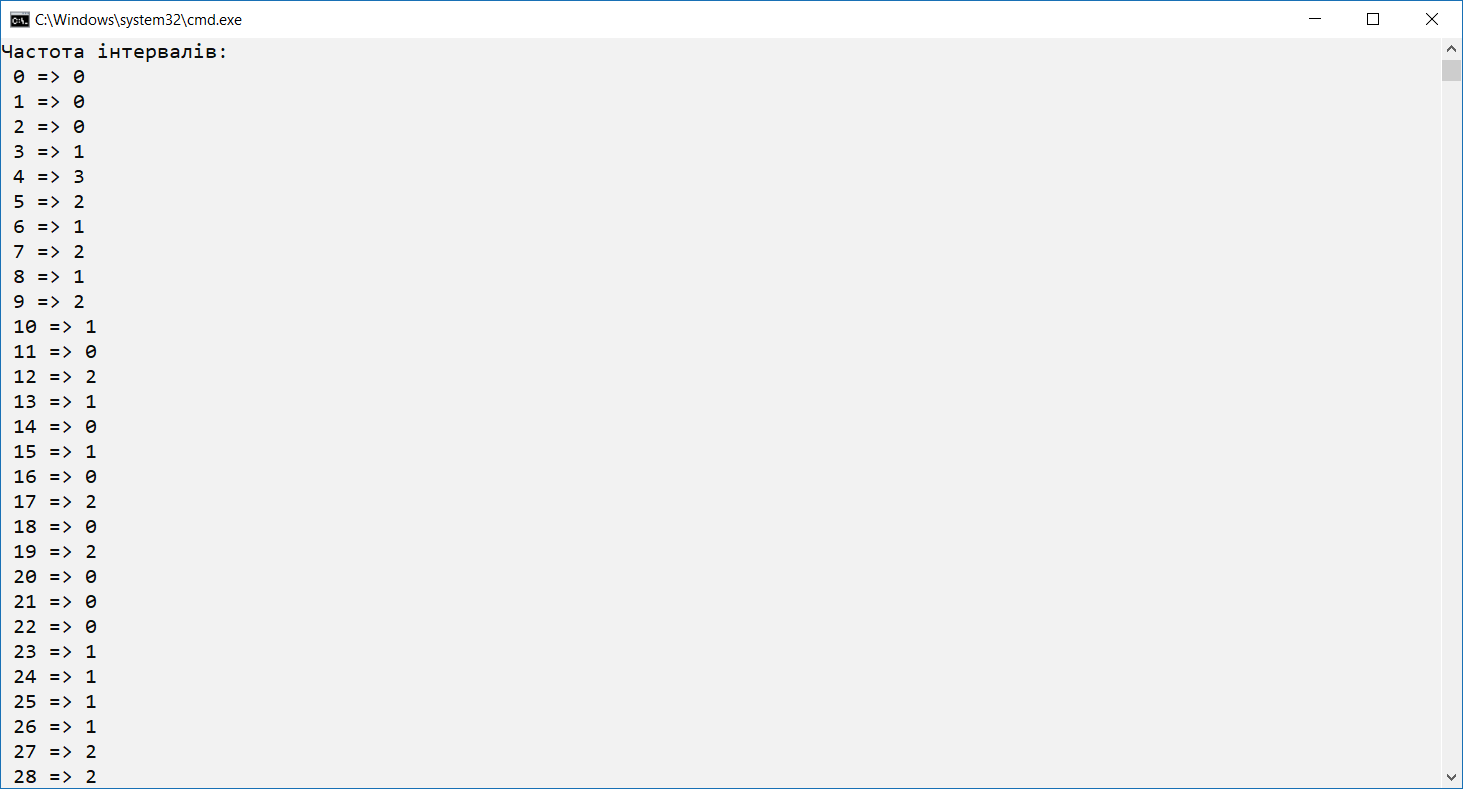


Рисунок 2.2 - Частота інтервалів

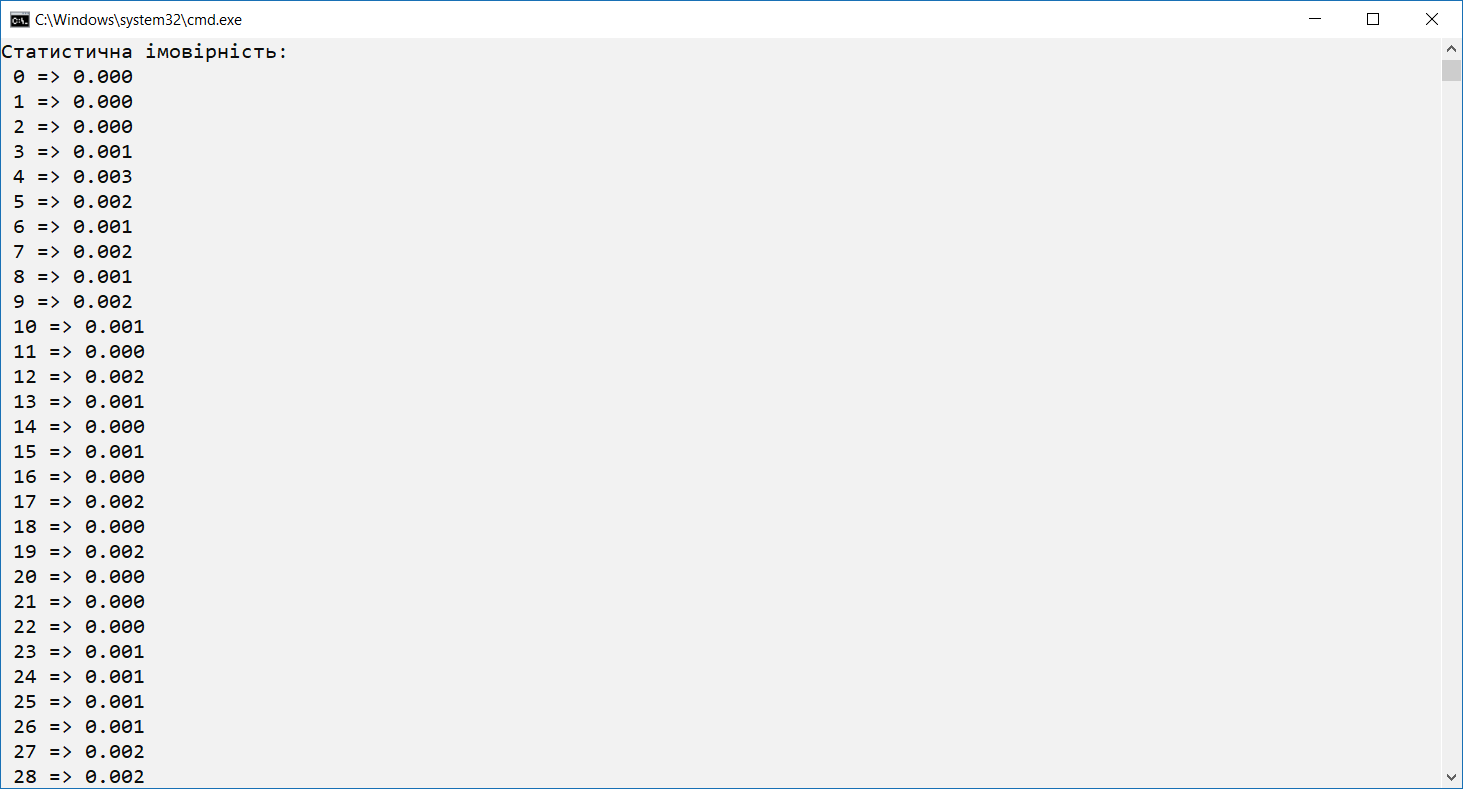


Рисунок 2.3 - Статична імовірність

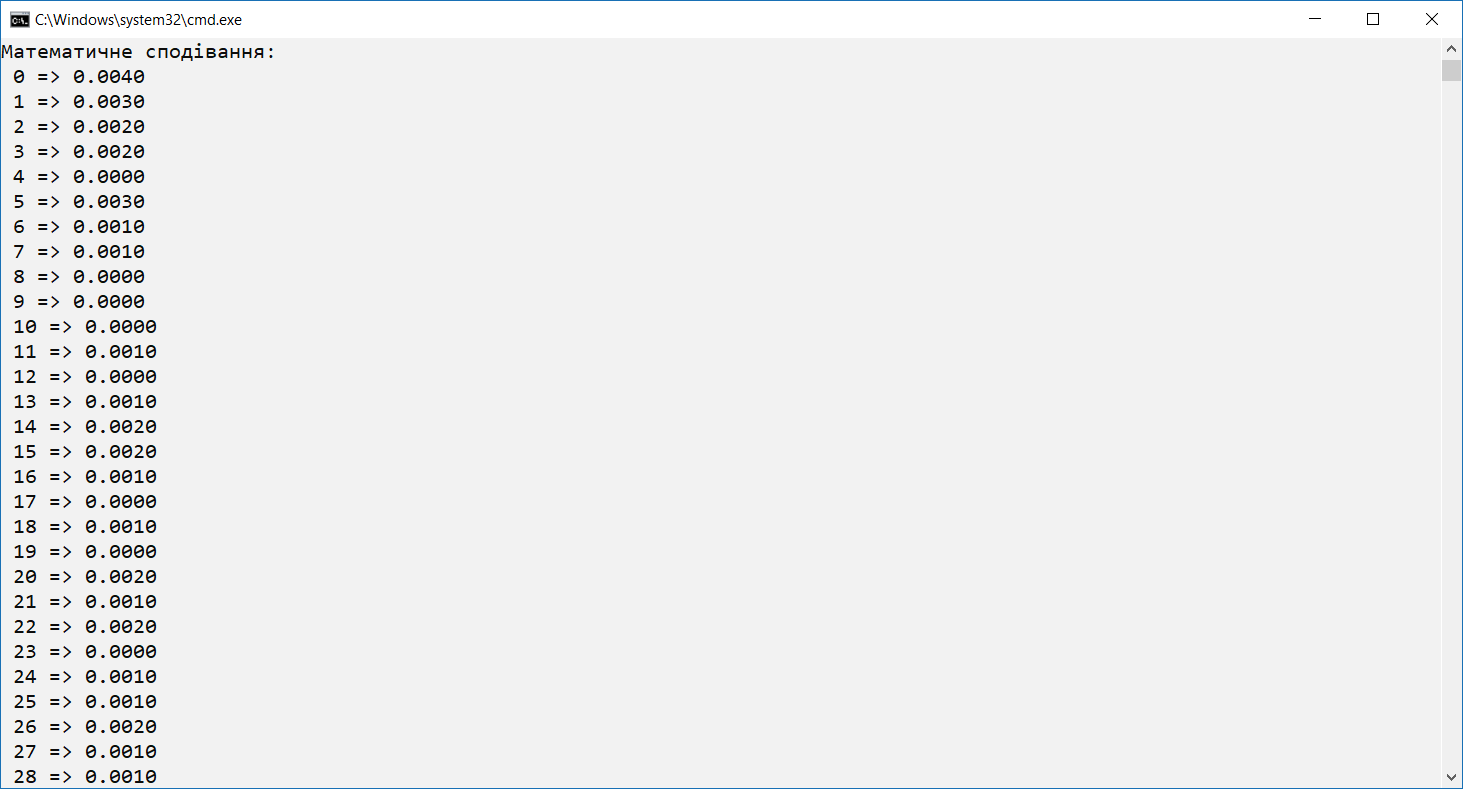




Рисунок 2.4 - Математичне сподівання

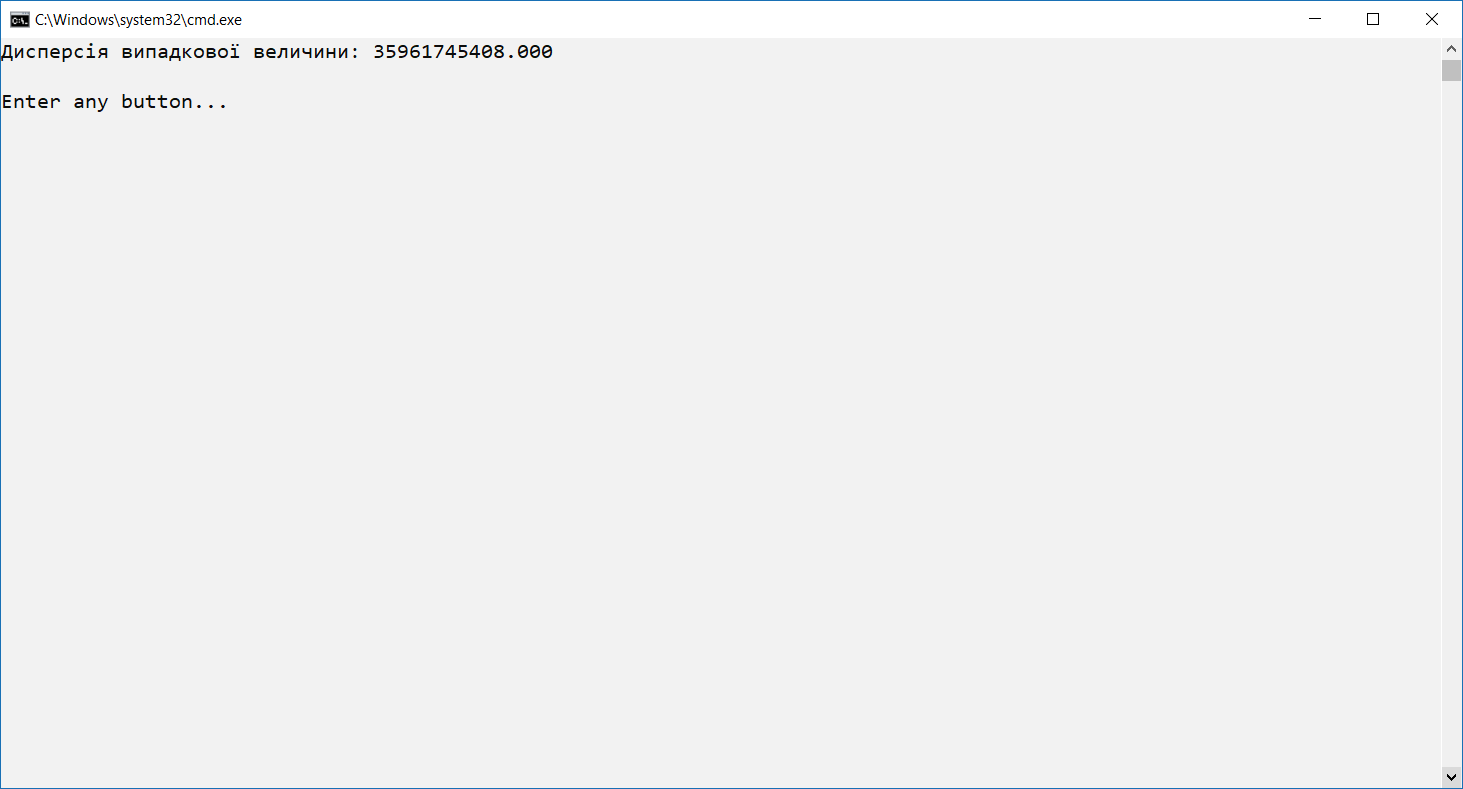


Рисунок 2.5 - Дисперсія випадкової величини



Рисунок 2.6 - Середньоквадратичне відхилення

***Висновки:*** на лабораторній роботі було ознайомлено з конгруентним методом генерування псевдовипадкових чисел, а також формуванням та обробкою масивів даних; дізналися, що числа не можуть бути випадковими, так як розраховуються за алгоритмом, через що будуть повторюватися. Випадковою вважають послідовність числа Пі.

ЛАБОРАТОРНА РОБОТА № 3

**Зв’язний список, стек, черга**

***Мета:*** ознайомитися з основами роботи з двозв’язним списком, однозв’язним

списком, стеком та чергою. Розробити основні функції.

**Хід роботи:**

**Завдання 1**: Розробити всі основні функції роботи з двозв’язним списком та однозв’язним списком (стеком та чергою). Розробити програму роботи з двозв’язним списком, однозв’язним списком.

Зв’язаний список – це структура, яка дозволяє нам динамічно додавати та забирати елементи. Великий недолік зв’язаного списку – доступ до н-го елементу, адже нам потрібно буде пройтися по більшій частині списку. В масиві такої проблеми немає, адже всі елементи знаходяться один за одним.

***List***

Створимо дві структури, які будуть відповідати за зберігання певного елементу і усього списку:

typedef int type;

struct elem

{

type value;

struct elem\* next;

struct elem\* prev;

};

struct myList

{

struct elem\* head;

struct elem\* tail;

int size;

};

В структурі елементу зберігається інформація та показники на наступний та попередній елемент (двозв’язний список).

Наступна функція створює та повертає показник на новий список:

List\* createList(void)

{

List\* list = (List\*)malloc(sizeof(List));

if (list) {

list->size = 0;

list->head = list->tail = NULL;

}

else

printf("\nError, can't create a list!!!\n");

return list;

}

Функція **deleteList** видаляє список.

void deleteList(List \*list) {

elem\* head = list->head;

elem\* next = NULL;

while (head) {

next = head->next;

free(head);

head = next;

}

free(list);

list = NULL;

}

Функція **pushFront** перевіряє чи створили ми список, потім створює новий елемент де встановлює потрібні дані, показник на наступний елемент, та змінює показник на вершину в списку.

int pushFront(List\* list, type\* data) {

if (!list)

{

printf("\nFirstly create a list!!!\n");

return -1;

}

elem\* node = (elem\*)malloc(sizeof(elem));

if (!node) {

printf("\nError, can't create a node!!!\n");

return -1;

}

node->value = \*data;

node->next = list->head;

node->prev = NULL;

if (!isEmptyList(list)) {

list->head->prev = node;

}

else {

list->tail = node;

}

list->head = node;

list->size++;

return 0;

}

Функція **popFront** знищує елемент якій знаходиться в передній частині списку та повертає дані, які там зберігалися.

int popFront(List\* list, type\* data) {

if (!list || isEmptyList(list))

{

printf("\nFirstly create a list!!!\n");

return -1;

}

elem\* node;

node = list->head;

list->head = list->head->next;

if (!isEmptyList(list)) {

list->head->prev = NULL;

}

else {

list->tail = NULL;

}

\*data = node->value;

list->size--;

free(node);

return 0;

}

Функції **pushBack** та **popBack** працюють аналогічно але вже з кінцем списку.

int pushBack(List\* list, type\* data) {

if (!list) {

printf("\nFirstly create a list!!!\n");

return -1;

}

elem\* node = (elem\*)malloc(sizeof(elem));

if (!node) {

printf("\nError, can't create a node!!!\n");

return -1;

}

node->value = \*data;

node->next = NULL;

node->prev = list->tail;

if (!isEmptyList(list)) {

list->tail->next = node;

}

else {

list->head = node;

}

list->tail = node;

list->size++;

return 0;

}

int popBack(List\* list, type\* data) {

if (!list || isEmptyList(list))

{

printf("\nFirstly create a list!!!\n");

return -1;

}

elem \*node = NULL;

if (isEmptyList(list)) {

return -1;

}

node = list->tail;

list->tail = list->tail->prev;

if (!isEmptyList(list)) {

list->tail->next = NULL;

}

else {

list->head = NULL;

}

\*data = node->value;

list->size--;

free(node);

return 0;

}

Функція **getNode** повертає елемент за вказаним індексом, вона проходить по частині списку, якщо елементів буде багато – це може сильно вплинути на час виконання оперіції

elem\* getNode(List\* list, int index) {

elem \*node = NULL;

int i;

if (index >= list->size || index < 0) {

return (NULL);

}

if (index < list->size / 2) {

i = 0;

node = list->head;

while (node && i < index) {

node = node->next;

i++;

}

}

else {

i = list->size - 1;

node = list->tail;

while (node && i > index) {

node = node->prev;

i--;

}

}

printf("\n%d) %d\n", index, node->value);

return node;

}

Функція **printList** виводить список на консоль.

int printList(List\* list) {

if (list == NULL)

{

printf("\nCreate list firstly!!!\n");

return -1;

}

elem\* node = list->head;

if (isEmptyList(list)) {

printf("\nError, list is empty!!!\n");

return -1;

}

for (int i = 0; node; i++) {

printf("\n%d) %d\n", i + 1, node->value);

node = node->next;

}

}

В кінці реалізовуємо меню для вибору функцій та пишемо функцію main.

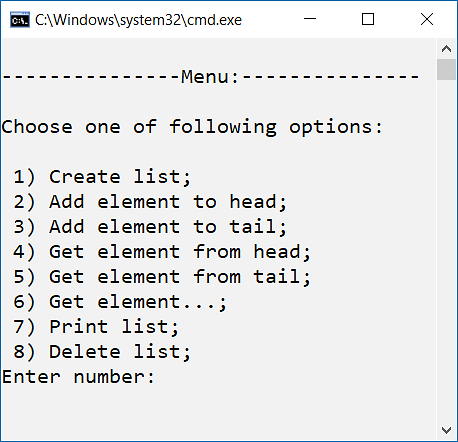


Рисунок 3.1 - Головне меню

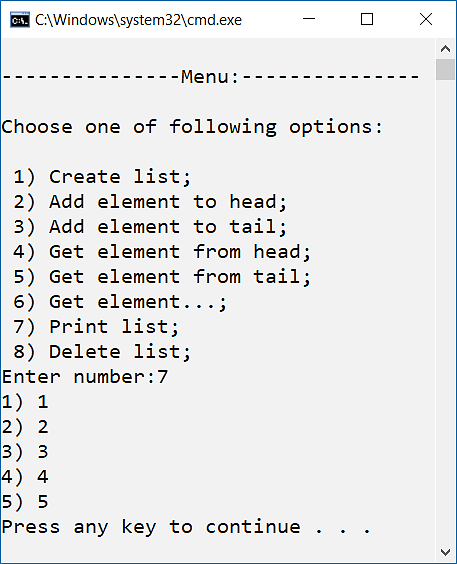


Рисунок 3.2 - Виведення списку

Аналогічно реалізовуємо функції для стека.

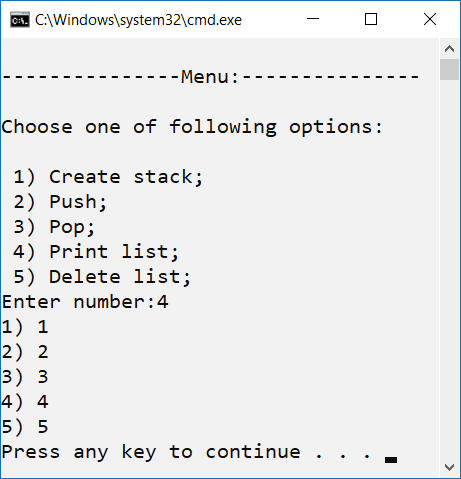


Рисунок 3.3 - Виведення стека

***Висновки:*** на даній лабораторній роботі навчилися створювати власні структури даних, такі як двозв’язні списки, стеки, черги. У кожного з них є свої плюси та мінуси. Наприклад для динамічного змінювання кількості елементів краще використовувати списки, але якщо постійно потрібно отримувати доступ до елементів, то краще використовувати масив. В масиві всі елементи розташовані в купі один за одним, коли в списку елементи розташовані в різних частинах купи.

ЛАБОРАТОРНА РОБОТА № 4

**Методи чисельного інтегрування**

***Мета:*** вивчення найпростіших квадратних формул (формули прямокутників, формули трапецій та формули Сімпсона) і метода Монте-Карло

**Хід роботи:**

**Завдання 1**: Розробити програму обчислення інтегралу функції по квадратній формулі та методу Монте-Карло.

Так як комп’ютер не може обчислити інтеграл функції, нам потрібно обрати інший спосіб обчислити площу складної фігури. Одним з таких способів є використання методу прямокутників. Ми ділимо фігуру на однакові за шириною прямокутники після чого знаходимо суму їх площ. Цей спосіб не дуже точний, адже не може повністю повторити контур фігури, особливо при великій ширині прямокутників.

double x = a, sRect = 0;

for (int i = 0; i < n; i++, x += h)

{

sRect += (-Math.Pow(x, 2) + c) \* h;

}

Наступний за ефективністю іде метод трапецій. Різницею від попереднього методу є те що ми ділимо фігуру не на прямокутники а на трапеції, що вже дає приріст в точності.

x = a;

double sTrap = 0;

for (int i = 0; i < n; i++, x += h)

{

sTrap += (-Math.Pow(x, 2) + c) + (-Math.Pow(x + h, 2) + c);

}

sTrap \*= h / 2;

Далі іде формула Сімпсона. Тепер замість того щоб з’єднувати точки простими лініями ми будемо проводити арку через кожні 3 точки, що набагато точніше.

double sum1 = 0, sum2 = 0;

for (int i = 1; i < n / 2; i++)

{ sum1 += (-Math.Pow(a + (i \* 2 - 1) \* h, 2) + c);

if (i < n / 2 - 1)

sum2 += (-Math.Pow((a + i \* 2 \* h), 2) + c);

}

sum1 \*= 4;

sum2 \*= 2;

double sSimps = h / 3 \* (-Math.Pow(a, 2) - Math.Pow(b, 2) + c \* 2 + sum1 + sum2);

Також існує метод Монте-Карло, який застосовується не тільки для знаходження площі складних фігур, а й для визначенні ймовірності подій. Він базується на теорії відносності. Ми вписуємо нашу фігуру в прямокутник, після чого починаємо генерувати випадкові точки в діапазоні нашого прямокутника, чим їх більше, тим точніше буде результат. Далі підраховуємо кількість точок, які потрапили в нашу фігуру.

Метод стверджує, що площа нашої фігури дорівнює площі прямокутника помноженої на ймовірність попадання точок.

double D = Math.Abs(a - b) \* c, y;

Random rand = new Random();

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

{

y = rand.NextDouble() \* c;

x = rand.NextDouble() \* (b - a) + a;

if (y <= (-Math.Pow(x, 2) + c))

N++;

}

double sMC = (double)D \* (double)N / n;

Як приклад, взято площу функції f(x) = -x2 обмеженої лініями y = c, x = 2, x = 1.

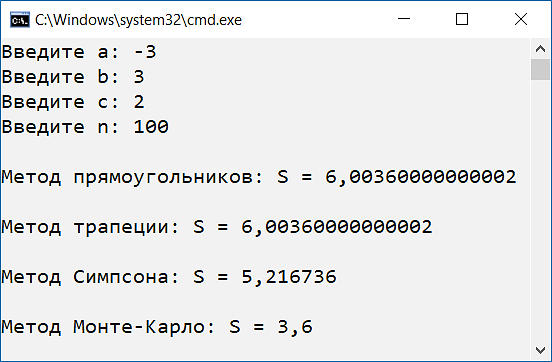


Рисунок 4.1 - Результат обрахунків при n = 100

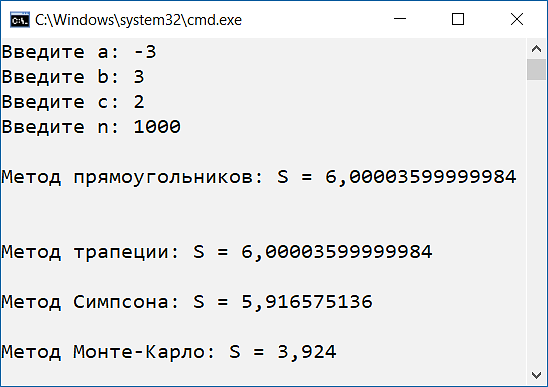


Рисунок 4.2 - Результат обрахунків при n = 1000

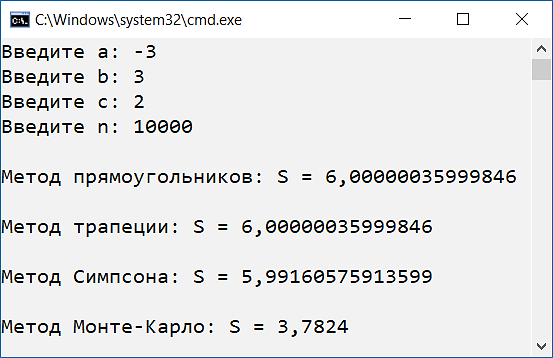


Рисунок 4.3 - Результат обрахунків при n= 10000

***Висновки:*** на даній лабораторній роботі ми дізналися про методи знаходження площі складних фігур, а саме методи: прямокутника, трапеції, Сімпсона, Монте-Карло. Методами прямокутників та трапецій ми можемо отримати досить точний результат навіть при маленькому значенні N. Метод Монте Карло дуже корисний коли потрібно отримати приблизні результати не виконуючи ніяких експериментів, тому він досить поширений, але для нього потрібна велика кількість точок, в іншому випадку результат буде не точним.

ЛАБОРАТОРНА РОБОТА № 5

**Методи сортування**

***Мета:*** реалізація алгоритмів сортування та дослідження їхніх характеристик.

**Хід роботи:**

**Завдання 1**: Реализовать алгоритмы сортировки:

a) сортировка выбором (структура данных - массив);

б) сортировка вставками (структура данных - массив);

в) сортировка вставками (структура данных – двусвязный список);

г) сортировка Шелла (структура данных - массив, вариант Р.Седжвика).

а) Сортуючи вибором N – 1 разів проходимо по невідсортованій частині масиву, знаходячи найменший елемент та переставляючи його в початок. При цьому метод не стійкий, адже ми схильні до переставляння однакових елементів

for (int i = 0, min, minInd = -1, temp; i < size - 1; i++)

{

min = int.MaxValue;

for (int j = i; j < size; j++)

if (arr[j] < min)

{

min = arr[j];

minInd = j;

}

temp = arr[i];

arr[i] = arr[minInd];

arr[minInd] = temp;

}

б) Сортуючи вставками – ми встаємо в кінець відсортованої частини, беремо наступний елемент і по черзі порівнюємо з попередніми поки не знайдемо менший елемент (бульбашкове сортування). На відміну від попереднього методу, на i-му кроці елементи в відсортованій частині можуть не знаходитися на своїй кінцевій позиції.

for (int i = 1; i < size; i++)

for (int j = i, temp; j > 0; j--)

{

if (arr1[j] < arr1[j - 1])

{

temp = arr1[j];

arr1[j] = arr1[j - 1];

arr1[j - 1] = temp;

}

else

break;

}

в) Використовуємо сортування вставками для сортування списків. Головна відмінність : замість того, щоб постійно міняти елементи між собою, ми виштовхуємо потрібний елемент, порівнюємо, та вставляємо вже в потрібне місце.

for (int i = 1, temp; i < size; i++)

{

temp = list[i];

list.RemoveAt(i);

for (int j = i; j > 0; j--)

{

if ((j - 1) == 0 && temp < list[0])

{

list.Insert(0, temp);

break;

}

if (temp >= list[j - 1])

{

list.Insert(j, temp);

break;

}

}

}

г) Алгоритм сортування Шелла базується на тому, що ми ділимо масив на проміжки, в яких сортуємо значення, потім зменшуємо ці проміжки поки вони не стануть розміром 1. Є багато методів для визначення цих проміжків, тут використовувався метод Р. Седжвика.

List<int> steps = ShellStepCalculator(arr2.Length, out int listSize);

for (int step; listSize >= 0; listSize--)

{

step = steps[listSize];

for (int i = step, temp; i < arr2.Length; i++)

{

for (int j = i; j - step >= 0 && arr2[j - step] > arr2[j]; j -= step)

{

temp = arr2[j - step];

arr2[j - step] = arr2[j];

arr2[j] = temp;

}

}

}

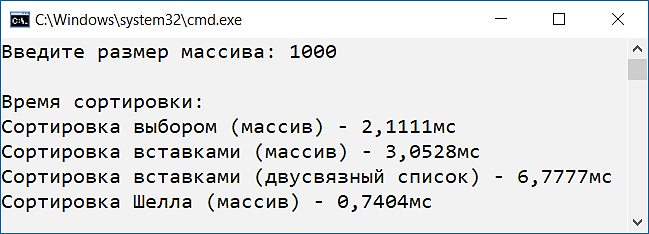


Рисунок 5.1 – Результат сортировки 1000 елементів

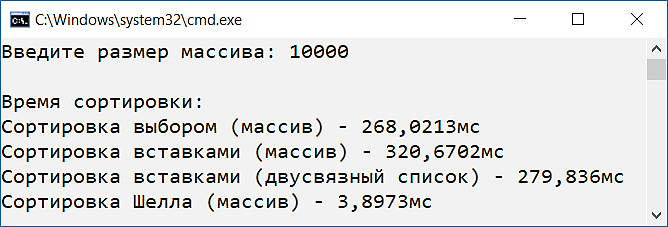


Рисунок 5.2 - Результат сортировки 10000 елементів

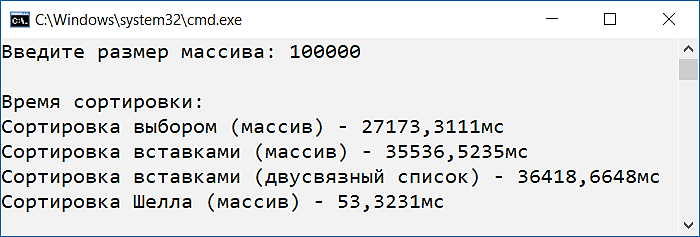


Рисунок 5.3 - Результат сортировки 100000 елементів

Таблица 1

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм / Час (мс) | 1000 елементів | 10000 елементів | 100000 елементів |
| сортировка выбором | 2.1 | 268.02 | 27 173.31 |
| сортировка вставками | 3.05 | 320.67 | 35 536.52 |
| сортировка вставками (двозв’язний список) | 6.78 | 279.83 | 36 418.67 |
| сортировка Шелла | 0.74 | 3.9 | 53.32 |

***Висновки:*** на даній лабораторній роботі було реалізовано алгоритми сортування та досліджено їхні характеристики; дізналися, що існує багато алгоритмів сортування, їхня ефективність досить сильно залежить від використання правильної структур даних. В даній лабораторній роботі було перевірено 3 алгоритми: вибором, вставками та Шелла. За отриманими даними найшвидшим з них виявився алгоритм Шелла.

ЛАБОРАТОРНА РОБОТА № 6

**Графи. Дерева. Алгоритми пошуку в довжину та в ширину**

***Мета:*** оволодіти та закріпити прийоми роботи з даними різного типу, організованих у вигляді дерев та їх частинного випадку – бінарного дерева. Отримати практичні навички роботи з деревами.

**Хід роботи:**

**Завдання 1**: Реалізувати алгоритми DFS та BFS.

DFS (Depth First Search) – це один з алгоритмів пошуку або подорожі по графу. Суть алгоритму полягає в тому, щоб знайти потрібну вершину подорожуючи по одній гілці глибину, якщо ми доходимо до вершин, які вже не мають нащадків, то ми повертаємося до батьківського елементу та шукаємо наступного нащадка.

public bool DFS(int startVerticeId, int endVerticeId)

{

Stack<int> visited = new Stack<int>();

visited.Push(startVerticeId);

for (int i = startVerticeId; i < Adjacency.GetLength(0);)

{

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

{

if (j < Adjacency.GetLength(0))

{

if (Adjacency[i, j] != 0)

{

if (j == endVerticeId)

{

visited.Push(j);

displayPath(visited);

return true;

}

else if (!visited.Contains(j))

{

visited.Push(j);

i = j;

break;

}

}

}

else

{

if (visited.Count == 1)

return false;

j = visited.Pop();

i = visited.First();

}

}

}

return false;

}

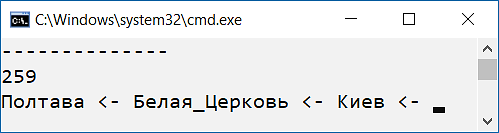


Рисунок 6.1 - Результат роботи алгоритму DFS

BFS (Breadth First Search) – це також алгоритм пошуку або подорожі, але на цей раз ми не йдемо в глибину по одній гілці, а проходимо кожен рівень графу по черзі, при цьому знаходячи всіх нащадків та додаючи їх в чергу.

public void BFS(int startVerticeId, int endVerticeId)

{

Stack<int> visited = new Stack<int>();

Queue<int> childs = new Queue<int>();

childs.Enqueue(startVerticeId);

while (childs.Count != 0)

{

var vertice = childs.Dequeue();

Console.Write(Enum.GetName(typeof(Cities), vertice) + " -> ");

visited.Push(vertice);

var verticeChilds = getChilds(vertice);

foreach (var child in verticeChilds)

{

if (visited.Contains(child))

continue;

childs.Enqueue(child);

Console.Write(Enum.GetName(typeof(Cities), child) + "; ");

}

Console.WriteLine("");

}

}

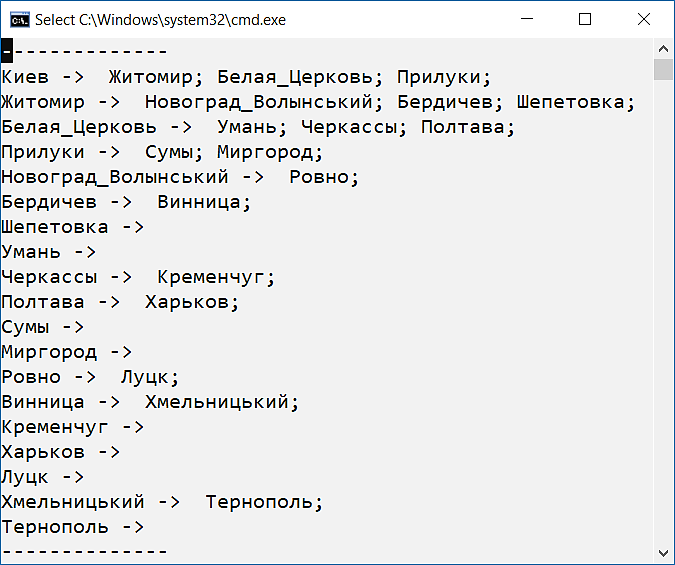


Рисунок 6.2 - Результат роботи алгоритму BFS

***Висновки:*** на даній лабораторній роботі оволоділи та закріпили прийоми роботи з даними різного типу, організованих у вигляді дерев та їх частинного випадку – бінарного дерева. Отримано практичних навичок роботи з деревами.

Отже, два алгоритми, які були щойно розглянуті виконують однакові функції, але використовуються в різних випадках. Метод, який буде краще в конкретному випадку, залежить від глибини, ширини та частоти зустрічі з цілю.

На приклад:

Якщо знаємо, що рішення знаходиться недалеко від кореня дерева, більш широкий пошук по ширині (BFS) може бути краще.

Якщо дерево дуже глибоке і рішення рідкісні пошук в глибину (DFS) може зайняти дуже багато часу, в той час як BFS буде ефективніше.

Якщо дерево дуже широке, BFS може знадобитися занадто багато пам'яті, тому може бути абсолютно непрактичним.