Житомирський державний технологічний університет

Факультет інформаційно-комп’ютерних технологій

Кафедра комп'ютерних наук

**Звіт з лабораторних робіт**

**з предмету**

**«Алгоритми та структури даних»**

студента І курсу групи КН-19-2

спеціальності 122 «Комп'ютерні науки»

Копилова Владислава Руслановича

Житомир – 2020

ЗМІСТ

[**Лабораторна робота №1** 3](#_Toc40901758)

[**Лабораторна робота №2** 8](#_Toc40901759)

[**Лабораторна робота №3** 12](#_Toc40901760)

[**Лабораторна робота №4** 21](#_Toc40901761)

[**Лабораторна робота №5** 26](#_Toc40901762)

[**Лабораторна робота №6-7** 31](#_Toc40901763)

# **Лабораторна робота №1**

Робота з базовими типами даних

**Ціль роботи:** отримати практичні навички по роботі з базовими типами даних.

**Хід роботи**

**1.**

В цьому завданні була використана структура, яка містить бітові поля. Це забезпечить програмі використовувати значно менше пам’яті, адже ми самі зазначаємо скільки пам’яті буде використовуватися. Ми використовуємо пам’ять, яка обов’язково буде використовуватись, а іншу можна використати для занесення в іншу змінну.

struct Time {

unsigned short nHours : 5;

unsigned short nMin : 6;

unsigned short nSec : 6;

unsigned short nDay : 5;

unsigned short nMonth : 5;

unsigned short nYear : 6;

}time;

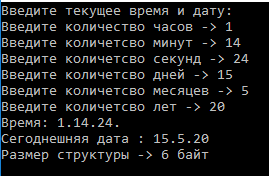


Рис.1.1.Результат

Якщо б ми ввели значну більшість кількість даних, то у структуру було б записано тільки ту кількість пам’яті, яку ми виділили у бітових полях.

**2.**

У цьому завданні ми використовуємо об’єднання для визначення знаку числа типу signed char, яке вводиться. Знак числа буде записуватися в 15 біт, тому за допомогою об’єднання ми поділимо число на 16 біт.

union number {

struct n {

unsigned short value0 : 1;

unsigned short value1 : 1;

unsigned short value2 : 1;

unsigned short value3 : 1;

unsigned short value4 : 1;

unsigned short value5 : 1;

unsigned short value6 : 1;

unsigned short value7 : 1;

unsigned short value8 : 1;

unsigned short value9 : 1;

unsigned short value10 : 1;

unsigned short value11 : 1;

unsigned short value12 : 1;

unsigned short value13 : 1;

unsigned short value14 : 1;

unsigned short value15 : 1;

}numbr;

signed short count;

}num;



Рис.1.2.1.Результат



Рис.1.2.2.Результат

**3.**

а) Переповнення даних типу signed char. Якщо значення преповнює максимально допустиме, то виконується кільцування та запис значення почнеться з іншої сторони. В даному прикладі максимальне значення 127, але всього значень 256, 132-256 = -124;

б) 210 = 00102, 310 = 00112 = -1

в) -1202 = -100010002, 3410 = 001000102

011001102 = 10210

г) 510 = 0000 01012 ;

Зробимо інверсію : 1111 10102 ;

Дадаємо одиниці: 1111 10112

Виходить : 1111 10112 = 25110

д) & - логічна операція множення.

Операція повертає 1, якщо обидва розряди 1;

Операція повертає 0, якщо обидва розряди 0 або вони різні.

5610 = 0011 10002

3810 = 0010 01102

0011 10002 & 0010 01102 = 0010 00002

0010 00002 = 3210

е) | - логічна операція додавання

Операція повертає 0, якщо обидва розряди 0,

Операція повертає 1, якщо обидва розряди 1 або вони різні.

5610 = 0011 10002

3810 = 0010 01102

0011 10002 | 0010 01102 = 0011 11102

0011 11102 = 6210

signed char value1\_1 = 5;

signed char value1\_2 = 127;

signed char value2\_1 = 2;

signed char value2\_2 = -3;

signed char value3\_1 = -120;

signed char value3\_2 = -34;

unsigned char value4 = -5;

signed char value5\_1 = 56;

signed char value5\_2 = 38;

signed char Result1 = value1\_1 + value1\_2;

signed char Result2 = value2\_1 + value2\_2;

signed char Result3 = value3\_1 + value3\_2;

printf("а) 5 + 127 = %d\n", Result1);

printf("б) 2-3 = %d\n", Result2);

printf("в) -120-34 = %d\n", Result3);

printf("г) (unsigned char)(-5) = %d\n", value4);

printf("д) 56 & 38 = %d\n", value5\_1 & value5\_2);

printf("е) 56 | 38 = %d\n", value5\_1 | value5\_2);

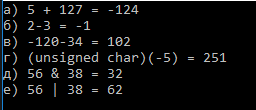


Рис.1.3.Результат

**4.**

В цьому завданні також використовується об’єднання. Ми поділимо число на 32 біти і на 4 байти, щоб вивести значення цього числа побітово та побайтово.

Для того, щоб вивести степінь потрібно відобразити 30-23 біти.

Мантиса збережена в 22-0 біті.

union numbers {

struct number {

unsigned short bit0 : 1;

unsigned short bit1 : 1;

unsigned short bit2 : 1;

unsigned short bit3 : 1;

unsigned short bit4 : 1;

unsigned short bit5 : 1;

unsigned short bit6 : 1;

unsigned short bit7 : 1;

unsigned short bit8 : 1;

unsigned short bit9 : 1;

unsigned short bit10 : 1;

unsigned short bit11 : 1;

unsigned short bit12 : 1;

unsigned short bit13 : 1;

unsigned short bit14 : 1;

unsigned short bit15 : 1;

unsigned short bit16 : 1;

unsigned short bit17 : 1;

unsigned short bit18 : 1;

unsigned short bit19 : 1;

unsigned short bit20 : 1;

unsigned short bit21 : 1;

unsigned short bit22 : 1;

unsigned short bit23 : 1;

unsigned short bit24 : 1;

unsigned short bit25 : 1;

unsigned short bit26 : 1;

unsigned short bit27 : 1;

unsigned short bit28 : 1;

unsigned short bit29 : 1;

unsigned short bit30 : 1;

unsigned short bit31 : 1;

}number;

struct num {

unsigned short byte1 : 8;

unsigned short byte2 : 8;

unsigned short byte3 : 8;

unsigned short byte4 : 8;

}bytes;

float value;

}num;

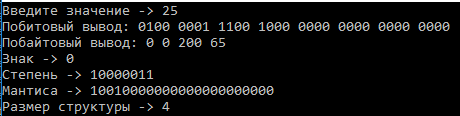


Рис.1.4.Результат

***Висновки :***  у ході виконання лабораторної роботи було виконано ознайомлення с структурами та об’єднаннями більш детально ніж раніше, я дізнався про те, як їх використовувати для більш раціонального та більш компактного запису даних, про те як розбивати значення числа побітово і побайтово у програмному коді та робити з ними операції на мові програмування C.

# **Лабораторна робота №2**

Генерування послідовності псевдовипадкових значень

**Ціль роботи:** ознайомитись з методами генерування випадкових чисел, а також формуванням та обробкою масивів даних.

**Хід роботи**

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.

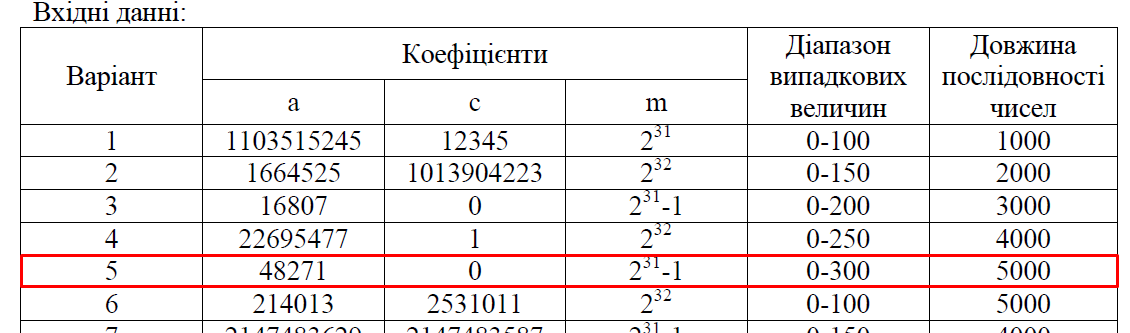
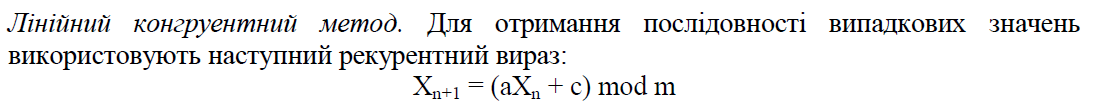


Рис.2.1.Варіант завдання

Генерації чисел за формулою лінійною конгруентною формулою:



class RandomNumberGenerator

{

private const long m = 2147483647; // aka 2^31-1

private const long a = 48271;

private const long c = 0;

private long x;

public RandomNumberGenerator()

{

Console.Write("x = ");

x = long.Parse(Console.ReadLine());

}

public RandomNumberGenerator(long x)

{

this.x = x;

}

public long Next()

{

x = ((a \* x) + c) % m;

return x;

}

public long Next(long maxValue)

{

return Next() % maxValue;

}

}

Генерація псевдовипадкових чисел у масив:

var arr = new double[5000];

var rng = new RandomNumberGenerator();

for (int i = 0; i < 5000; i++)

{

var rnd = rng.Next(300);

arr[rnd] += 1;

Console.WriteLine($"{i} -> {rnd} ");

}

Довжина послідовності згідно до варіанту:

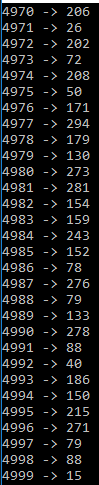


Рис.2.2.Результат

Для визначення частоти інтервалів та статистичної імовірність появи псевдопивадкових чисел буде використан наступний код:

var frequency = new double[300];

Console.WriteLine("Частота интервалов появления случайных чисел:");

int count = 0;

for (int i = 0; i < 300; i++)

{

frequency[i] = arr[i];

count++;

Console.WriteLine("Число {0} = {1}", i + 1, frequency[i]);

}

var stat = new double[300];

double count1 = 0;

for (int i = 0; i < 300; i++)

{

stat[i] = (frequency[i] / 5000);

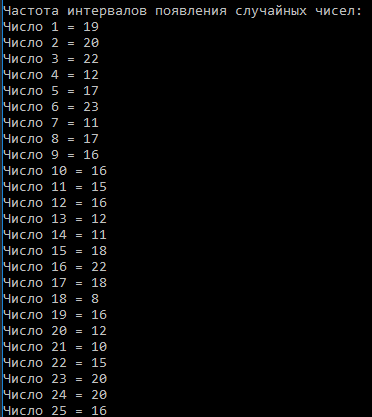
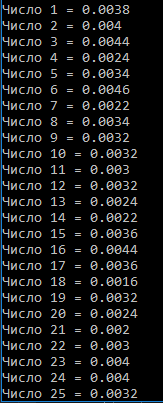
count1 += stat[i];

Console.WriteLine("Число {0} = {1}", i + 1, stat[i]);

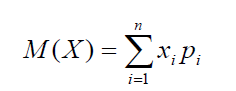
}

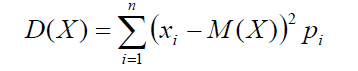
Console.WriteLine($"Частота интервалов появления случайных чисел: {count}");

Console.WriteLine($"Статистическая вероятность случайных чисел: {(count1)}");

  
Рис.2.3.Результат Рис.2.4.Результат

Для розрахування математичного сподівання, дисперсії та сердньоквадратичного відхилення випадкових величин ми скористуємось відповідними формулами







Та зробимо відповідний код до формул:

double mathExpectedValue = 0;

for (int i = 0; i < 300; i++)

{

mathExpectedValue += i \* stat[i];

}

Console.WriteLine($"Математическое ожидание: {mathExpectedValue:F3}");

double dispersion = 0;

for (int i = 0; i < 300; i++)

{

dispersion += Math.Pow((i - mathExpectedValue), 2) \* stat[i];

}

Console.WriteLine($"Дисперсия случайных величин: {dispersion:F3}");

Console.WriteLine($"Среднее квадратическое отклонение случайных величин: {Math.Sqrt(dispersion):F3}");

Console.ReadLine();

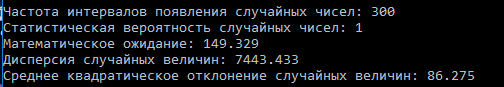


Рис.2.5.Результат

***Висновки:*** у ході виконання лабораторної роботи була вивчена програмна реалізація лінійного конгруентного методу для отримання послідовності випадкових чисел і характеристики до заданої послідовності, а саме математичне сподівання, дисперсія та середньоквадратичне відхилення.

# **Лабораторна робота №3**

Зв’язний список, стек, черга. Зворотній польський запис

**Ціль роботи:** ознайомитися з основами роботи з двозв’язним списком, однозв’язним списком, стеком та чергою. Розробити основні функції для обчислення арифметичного виразу, записаного з використанням зворотного польського запису.

**Хід роботи**

**1.**

Основні функції роботи з двозв’язним списком:

Дадавання елементу у список, елемент додається у початок списку:

public void Add(T data)

{

var item = new DoubleItem<T>(data);

DoubleItem<T> temp = Head;

item.Next = temp;

Head = item;

if (Count == 0)

{

Tail = Head;

}

else

{

temp.Previous = item;

}

Count++;

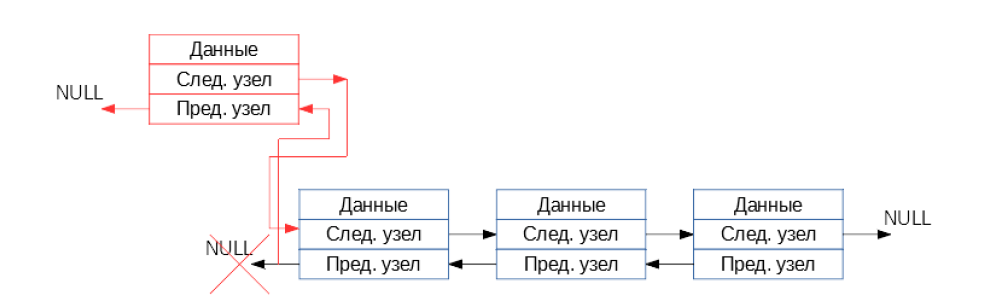
}

Рис.3.1.Графічне зображення додавання елемененту у початок списку

Додавання елементу у кінець списку:

public void AddToEnd(T data)

{

var item = new DoubleItem<T>(data);

if (Count == 0)

{

Head = item;

Tail = item;

Count = 1;

return;

}

Tail.Next = item;

item.Previous = Tail;

Tail = item;

Count++;

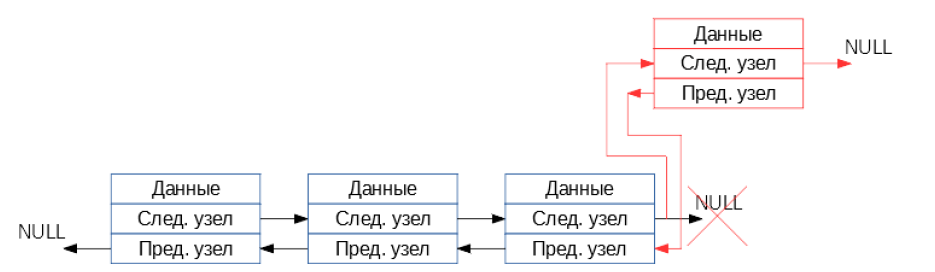
}

Рис.3.2.Графічне зображення додавання елементу у кінець списку

Видалення елементу зі списку:

public void Delete(T data)

{

var current = Head;

while (current != null)

{

if (current.Data.Equals(data))

{

break;

}

current = current.Next;

}

if (current != null)

{

if (current.Next != null)

{

current.Previous.Next = current.Previous;

}

else

{

Tail = current.Previous;

}

if (current.Previous != null)

{

current.Previous.Next = current.Next;

}

else

{

Head = current.Next;

}

Count--;

return;

}

return;

}

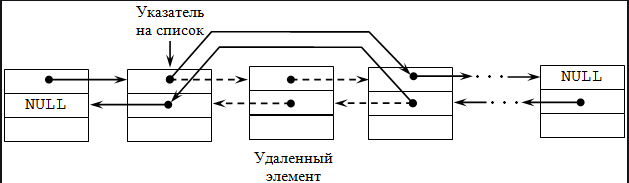


Рис.3.3.Графічне зображення видалення елемента зі списку

Очищення списку:

public void Clear()

{

Head = null;

Tail = null;

Count = 0;

}

Реверсування списку:

public DoubleLinkedList<T> Reverse()

{

var result = new DoubleLinkedList<T>();

var current = Tail;

while (current != null)

{

result.Add(current.Data);

current = current.Previous;

}

return result;

}

**2.**

Програма роботи з двозв’язним списком буде виконуватись у діалоговому вікні, тому було створено відповідне меню

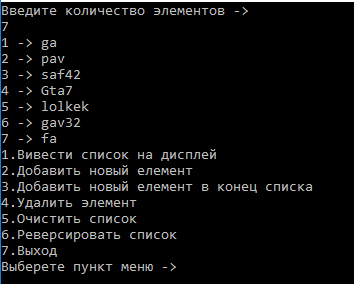


Рис.3.4.Результат

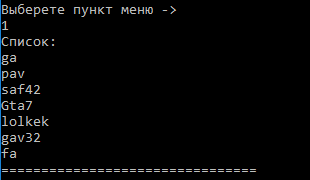


Рис.3.5.Виведення списку на екран

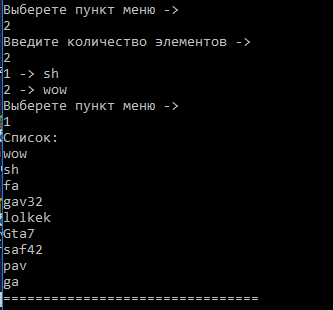


Рис.3.6.Додавання елементів у початок списку

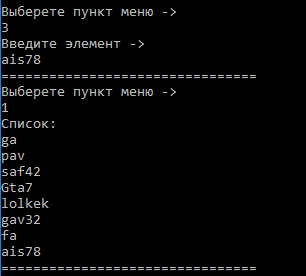


Рис.3.7.Дадавання елементу у кінець списку

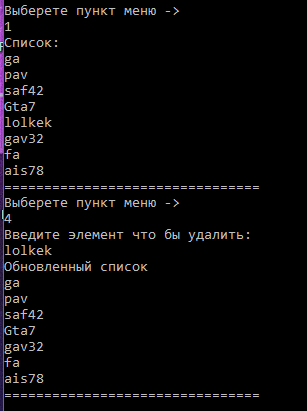


Рис.3.8.Видалення елементу зі списку

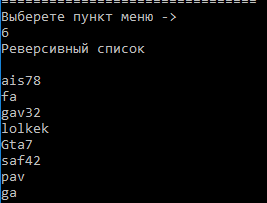


Рис.3.9.Реверсування списку

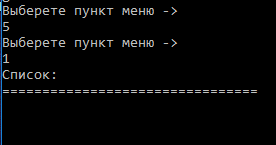


Рис.3.10.Очищення списку

**3.**

Зробимо окремий клас для реалізації алогоритму, в ньому будуть методи для роботи з виразом, який вводить користувач.

Цей метод конвертує строку, в якій буде написано зворотній польський запис.

Метод перетворює його в звичайний вираз, який компілятор буде в змозі підрахувати та вивести правильний результат

static private string Expression(string value)

{

string outputValue = string.Empty;

Stack<char> valueStack = new Stack<char>();

for (int i = 0; i < value.Length; i++)

{

if (Delimeter(value[i]))

continue;

if (Char.IsDigit(value[i]))

{

while (!Delimeter(value[i]) && !Operator(value[i]))

{

outputValue += value[i];

i++;

if (i == value.Length) break;

}

outputValue += " ";

i--;

}

if (Operator(value[i]))

{

if (value[i] == '(')

valueStack.Push(value[i]);

else if (value[i] == ')')

{

char s = valueStack.Pop();

while (s != '(')

{

outputValue += s.ToString() + ' ';

s = valueStack.Pop();

}

}

else

{

if (valueStack.Count > 0)

{

if (Priority(value[i]) <= Priority(valueStack.Peek()))

{

outputValue += valueStack.Pop().ToString() + " ";

}

}

valueStack.Push(char.Parse(value[i].ToString()));

}

}

}

while (valueStack.Count > 0)

{

outputValue += valueStack.Pop() + " ";

}

return outputValue;

}

Метод Counting виконує операції та виводить результат

static private double Counting(string value)

{

double result = 0;

Stack<double> temp = new Stack<double>();

for (int i = 0; i < value.Length; i++)

{

if (Char.IsDigit(value[i]))

{

string a = string.Empty;

while (!Delimeter(value[i]) && !Operator(value[i]))

{

a += value[i];

i++;

if (i == value.Length) break;

}

temp.Push(double.Parse(a));

i--;

}

else if (Operator(value[i]))

{

double a = temp.Pop();

double b = temp.Pop();

switch (value[i])

{

case '+': result = b + a;

break;

case '-': result = b - a;

break;

case '\*': result = b \* a;

break;

case '/': result = b / a;

break;

case '^': result = double.Parse(Math.Pow(double.Parse(b.ToString()), double.Parse(a.ToString())).ToString());

break;

}

temp.Push(result);

}

}

return temp.Peek();

}

Метод Delimetr аналізує символи, якщо символ є пробілом, то повертає true, якщо ні, то false

static private bool Delimeter(char symbol)

{

if ((" =".IndexOf(symbol) != -1))

return true;

return false;

}

Метод Operator аналізує символ, якщо символ є оператором, то повертає true, якщо ні, то false

static private bool Operator(char symbol)

{

if (("+-/\*^()".IndexOf(symbol) != -1))

return true;

return false;

}

Метод Priority визначає пріорітетність операції.

static private byte Priority(char operation)

{

switch (operation)

{

case '(': return 0;

case ')': return 1;

case '+': return 2;

case '-': return 3;

case '\*': return 4;

case '/': return 4;

case '^': return 5;

default: return 6;

}

}

Метод приймає вираз, який вже було перетворено в звичайний вигляд і передає його для підрахунку, в кінці повертає результат

static public double Calculate(string value)

{

string outputValue = Expression(value);

double Result = Counting(outputValue);

return Result;

}

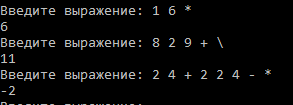


Рис.3.11. Результат

***Висновки:*** у ході виконання лабораторної роботи було ознайомлено з зв’язним списком, стеком, чергою та зворотнім польським записом. Я дізнався про особливості двохзв’язного списку, що це дуже добрий спосіб для зберігання інформації, адже усі дані зберігаються в неперервній області пам’яті, а такий спосіб зберігання пам’яті дозволяє програмі працювати швидше.

# **Лабораторна робота №4**

Методи чисельного інтегрування

**Ціль роботи:** вивчення найбільш простих квадратурних формул (формули

прямокутників, формули трапецій і формули Сімпсона) і методу Монте-Карло.

**Хід роботи**

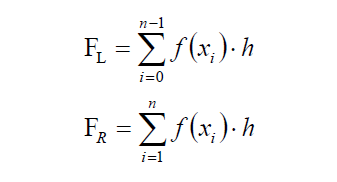
**1.**

Метод прямокутників

Було зроблено два методи прямокутників, а саме метод лівих та правих прямокутників.

Метод полягає в наближеному обчисленні площі під графіком, підсумовуванням площ кінцевого числа прямокутників, ширина яких буде визначатися відстанню між відповідними сусідніми вузлами інтегрування, а висота - значенням підінтегральної функції в цих вузлах.

Значення інтеграла обчислюється за формулами:



static double RightTriangle(double Result)

{

Console.WriteLine("Метод правых прямоугольников:");

double a, b, n;

double step = (b - a) / n;

double Summa = 0;

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

{

double x = a + i \* step;

Summa += Function(x);

}

Result = step \* Summa;

return Result;

}

static double LeftTriangle(double Result)

{

Console.WriteLine("Метод левых прямоугольников:");

double a, b, n;

double step = (b - a) / n;

double Summa = 0;

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

{

double x = a + i \* step;

Summa += Function(x);

}

Result = step \* Summa;

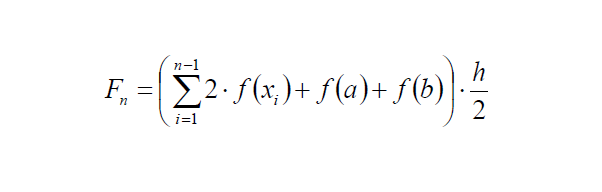
return Result;

}

**2.**

Метод трапеції

В основну ідею методу трапецій покладено заміну кривої підінтегральної функції на ламану. Ми розділяємо вказаний проміжок на рівні частини і сполучаємо лініями значення функцій на кінцях відрізку, тобто площу криволінійної трапеції наближено замінюємо на суму площ n трапецій.



static double Trapeze(double Result)

{

Console.WriteLine("Метод трапеции:");

double a, b, n;

double step = (b - a) / n;

double Summa = 0;

for (int i = 1; i <= n-1; i++)

{

double x = a + i \* step;

Summa += 2\*Function(x);

}

Result = (step / 2) \* (Summa + Function(a) + Function(b));

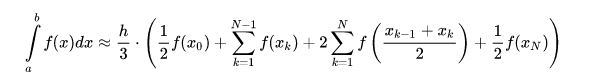
return Result;

}

**3.**

Метод Сімпсона

Суть методу полягає в наближенні підінтегральної функції на відрізку [a, b] інтерполяційним многочленом другого ступеня p2 (x), тобто наближення графіка функції на відрізку параболою. Для більш точного обчислення інтеграла, інтервал [a, b] розбивають на N = 2n елементарних відрізків однакової довжини і застосовують формулу Сімпсона на складових відрізках. Кожен складовою відрізок складається з сусідньої пари елементарних відрізків. Значення вихідного інтеграла є сумою результатів інтегрування на складових відрізках



static double SimpsonMethod(double Result)

{

Console.WriteLine("Метод трапеции:");

double a, b, n;

double step = (b - a) / n;

double Summa1 = 0;

double Summa2 = 0;

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

{

double x1 = a + i \* step;

if(i <= n - 1)

{

Summa1 += Function(x1);

}

double x2 = a + (i-1) \* step;

Summa2 += Function((x1 + x2)/2);

}

Result = (step / 3) \* (0.5\*Function(a) + Summa1 + 2\*Summa2 + 0.5\*Function(b));

return Result;

}

**4.**

Метод Монте-Карло

Ми приймаємо проміжок [a;b] і кількість точок, далі описуємо функцію прямокутником (частина функції де y <0 вказується як абсолютне значення, y мінімум завжди буде дорівнює 0. При попаданні за межі прямокутника максимальна межа розширюється в два рази від значення функції в цьому місці і починаємо спочатку.

static double MonteCarloMethod(double Result)

{

Console.WriteLine("Метод Монте-Карло:");

double a, b, n;

double total = 0;

double yMax = 0;

double x;

double funct;

int i = 0;

Random rnd = new Random();

do

{

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

funct = Math.Abs(Function(x));

if (yMax > funct)

{

total += funct;

i++;

}

else

{

yMax = funct \* 2;

i = 0;

}

} while (i < n);

Result = (b - a) \* total / n;

return Result;

}

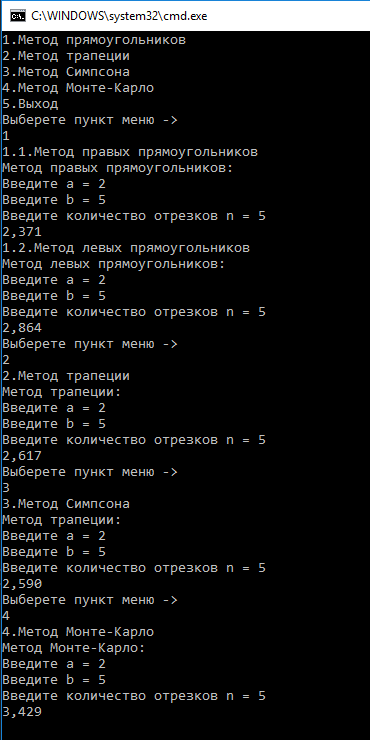
****

Рис.4.Результат

***Висновки:*** у ході виконання лабораторної роботи були опрацьовані методи численного інтегрування, а саме метод прямокутників (лівих та правих), метод трапеції, метод Сімпсона (метод парабол) та метод Монте-Карло. Усі методи мають більш-менш схожі результати, але метод Монте-Карло є найбільш універсальним, але в той же час не зовсім точним, адже це залежить від кількості згенерованих точок.

# **Лабораторна робота №5**

Методи сортування

**Ціль роботи:** реалізація алгоритмів сортування та дослідження їх характеристик (швидкодія, необхідний обсяг пам'яті, область застосування і т.д).

**Хід роботи**

Реалізувати алгоритми сортування:

а) сортування вибором

Ідея методу полягає в тому, щоб створювати відсортовану послідовність шляхом приєднання до неї одного елемента одним в правильному порядку.

Кроки алгоритму:

-знаходимо номер мінімального значення в поточному списку

-виробляємо обмін цього значення із значенням першої невідсортоване позиції (обмін не потрібний, якщо мінімальний елемент вже знаходиться на даній позиції)

-тепер сортуємо хвіст списку, виключивши з розгляду вже відсортовані елементи

static void SelectionSort(int[] arr)

{

int min, temp;

int length = arr.Length;

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

{

min = i;

for (int j = i + 1; j < length; j++)

{

if (arr[j] < arr[min])

{

min = j;

}

}

if (min != i)

{

temp = arr[i];

arr[i] = arr[min];

arr[min] = temp;

}

}

}

б) сортування вставками

Алгоритм сортування, в якому елементи вхідної послідовності проглядаються по одному, і кожен новий надійшов елемент розміщується в підходяще місце серед раніше упорядкованих елементів

Ми розділяємо масив на 2 колекції и порівнюємо числа до тих пір, поки друга колекція не буде порожньою.

static void InsertionSort(int[] arr)

{

for (int i = 0; i < arr.Length - 1; i++)

{

int min = i;

for (int j = i + 1; j < arr.Length; j++)

{

if (arr[j] < arr[min])

min = j;

}

int temp = arr[i];

arr[i] = arr[min];

arr[min] = temp;

}

}

в) сортування вставками двохзв’язного списку

Спочатку створюємо список. Було прийняте рішення ,щоб код програми одразу при генерації чисел у список, сортував його, тобто числа одразу будуть ставати на своє місце.

static DoubleLinkedList InsertionSortList(DoubleLinkedList listInsertion, DoubleLinkedList listInsertionNext)

{

DoubleLinkedList current;

if (listInsertion == null)

{

listInsertion = listInsertionNext;

}

else if (listInsertion.Data >= listInsertionNext.Data)

{

listInsertionNext.Next = listInsertion;

listInsertionNext.Next.Previous = listInsertionNext;

listInsertion = listInsertionNext;

}

else

{

current = listInsertion;

while (current.Next != null && current.Next.Data < listInsertionNext.Data)

{

current = current.Next;

}

listInsertionNext.Next = current.Next;

if (current.Next != null)

{

listInsertionNext.Next.Previous = listInsertionNext;

}

current.Next = listInsertionNext;

listInsertionNext.Previous = current;

}

return listInsertion;

}

г) Сортування Шелла за методом Седжвіка.

Сортування Шелла по суті, це модифікований варіант сортування вставками. Сортування виконується попарно. З кожною ітерацією масив сортується. В останній ітераціїї масив вже буде майже відсортований, саме тому цей метод виграє у часі звичайним метод вставками.

static void ShellsSort(int[] arr)

{

int step, i, j;

int[] inc = new int[40];

int p1 = 1, p2 = 1, p3 = 1;

int s = -1;

do

{

if (++s % 2 == 0)

inc[s] = 8 \* p1 - 6 \* p2 + 1;

else

{

inc[s] = 9 \* p1 - 9 \* p3 + 1;

p2 \*= 2;

p3 \*= 2;

}

p1 \*= 2;

} while (3 \* inc[s] < inc.Length);

s = s > 0 ? --s : 0;

while (s >= 0)

{

step = inc[s--];

for (i = step; i < arr.Length; i++)

{

int temp = arr[i];

for (j = i - step; (j >= 0) && (arr[j] > temp); j -= step)

arr[j + step] = arr[j];

arr[j + step] = temp;

}

}

}

Ми створюємо масив, заповнюємо його генерацією випадкових чисел та за допомогою простіру імен System.Diagnostic використовуємо функцію time для виміру часу сортування

Stopwatch checkTime;

int[] arrSelection = new int[2000];

Console.WriteLine("1.Сортировка выбором:");

checkTime = Stopwatch.StartNew();

for(int i =0; i< arrSelection.Length; i++)

{

Random rnd = new Random();

arrSelection[i] = rnd.Next(-100, 100);

SelectionSort(arrSelection);

}

checkTime.Stop();

Console.WriteLine($"Время сортировки выбором -> {checkTime.ElapsedMilliseconds:F3} мс\n");

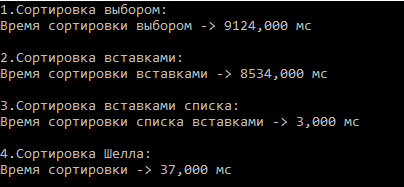


Рис.5.1.Результат

Можемо збільшити кількість елементів і заміряти час знову

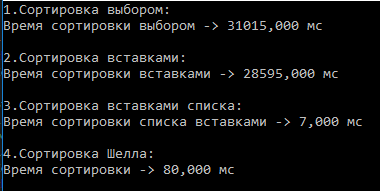


Рис.5.2.Результат

Тепер значно зменшимо кількість елементів

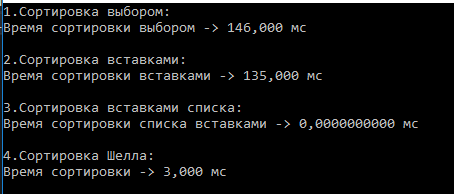


Рис.5.3.Результат

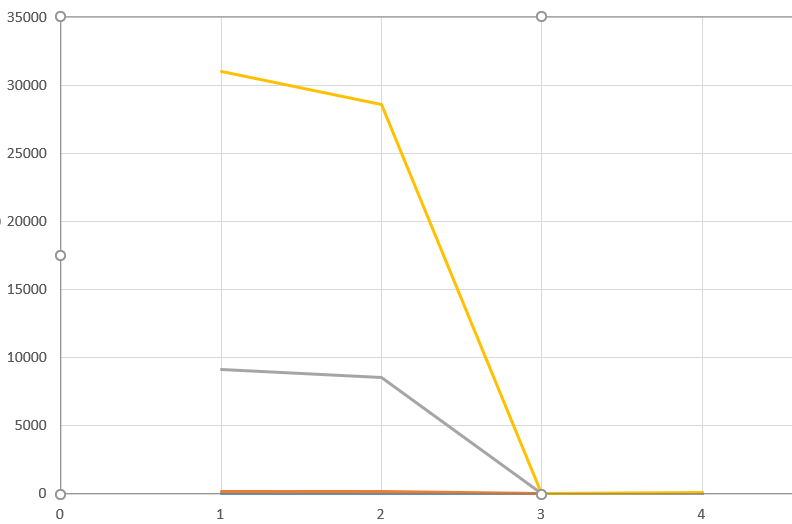


Рис.5.4.Графік

***Висновки :*** у ході виконання лабораторної роботи були використані та вивчені різні методи сортування такі, як бульбашкове сортування, сортування вставками та вибором, сортування Шелла. Останній метод сорування – сортування Шелла , є модифікованим методом вставок та він значно виграє у часі, що ми можемо побачити на графіку або у вимірах часу у результаті виконання програми. Метод вибором та вставками у часі майже однакові. За лабораторну роботу я зрозумів, що перші алгоритми є більш простими для розуміння, але менш ефективні у виконанні, коли в той час більш складний для розуміння метод Шелла є набагато кращий, бо швидше виконується.

# **Лабораторна робота №6-7**

Графи. Дерева. Алгоритми пошуку в глибину і в ширину

**Ціль роботи:** освоїти і закріпити прийоми роботи з даними різного типу, організованими у вигляді дерев та їх окремого випадку - бінарних дерев. Отримати практичні навички роботи з графами.

**Хід роботи**

Маршруты движения автобусов с Киева;

1. Киев –(135) Житомир –(80) Новоград-Волынский –(100) Ровно –(68) Луцк

2. Киев –(135) Житомир –(38) Бердичев –(73) Винница –(110) Хмельницкий –(104) Тернополь

3. Киев –(135) Житомир –(115) Шепетовка

4. Киев –(78) Белая церковь –(115) Умань

5. Киев –(78) Белая церковь –(146) Черкассы –(105) Кременчуг

6. Киев –(78) Белая церковь –(181) Полтава – (130) Харьков

7. Киев –(128) Прилуки –(175) Сумы

8. Киев –(128) Прилуки –(109) Миргород

Зобразимо ці маршрути на графіку, щоб було легше зробити матрицю сміжності для заданих маршрутів:

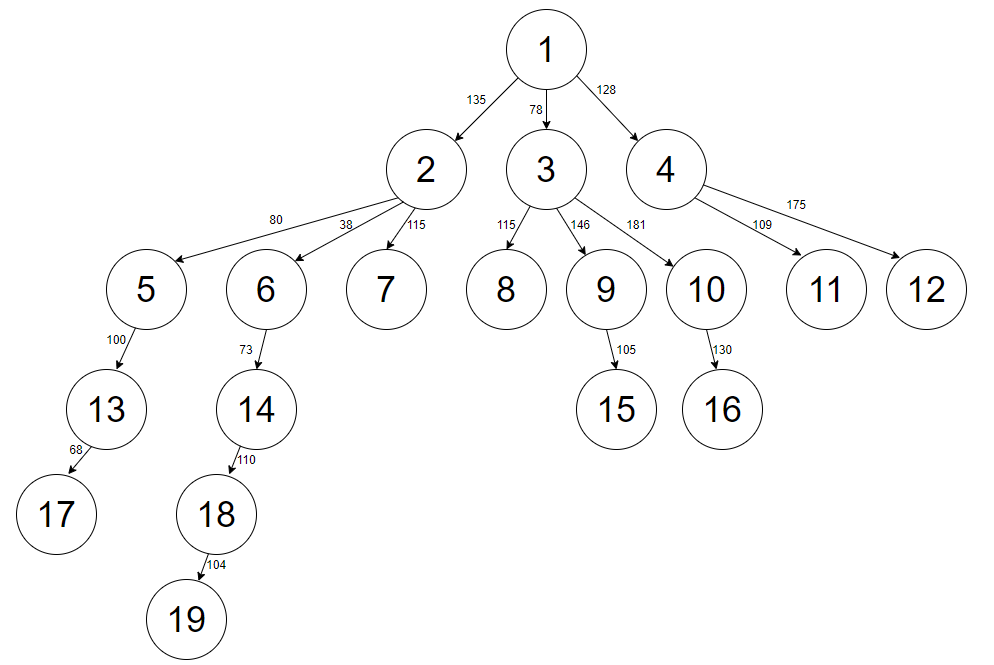


Рис.6.1.Графік

У програмному коді зробимо окремі класи для вершини, ребра та графа.

Потім у класі Program задамо шляхи, з якої вершини і в яку іде наш шлях, та за допомогою методів, зроблених в класі графа, зобразимо матрицю сміжності:

private static void GetMatrix(Graph graph)

{

Console.WriteLine("Матрица смежности:\n");

var matrix = graph.GetMatrix();

Console.Write("");

for (int i = 0; i < graph.VertexCount - 1; i++)

{

Console.Write($" {i + 1:D2} ");

}

Console.WriteLine();

Console.WriteLine(new string('\_', 172));

for (int i = 1; i < graph.VertexCount; i++)

{

Console.Write($"{i:D2}");

for (int j = 1; j < graph.VertexCount; j++)

{

if (matrix[i, j] == 0)

{

Console.Write($" | {matrix[i, j]} | ");

}

if (matrix[i, j] >= 100 && matrix[i, j] <= 999)

{

Console.ForegroundColor = ConsoleColor.DarkGray;

Console.Write($" | {matrix[i, j]} | ");

Console.ResetColor();

}

if (matrix[i, j] <= 99 && matrix[i, j] >= 10)

{

Console.ForegroundColor = ConsoleColor.DarkGray;

Console.Write($" | {matrix[i, j]} | ");

Console.ResetColor();

}

}

Console.WriteLine();

}

Console.WriteLine(new string('\_', 172));

}

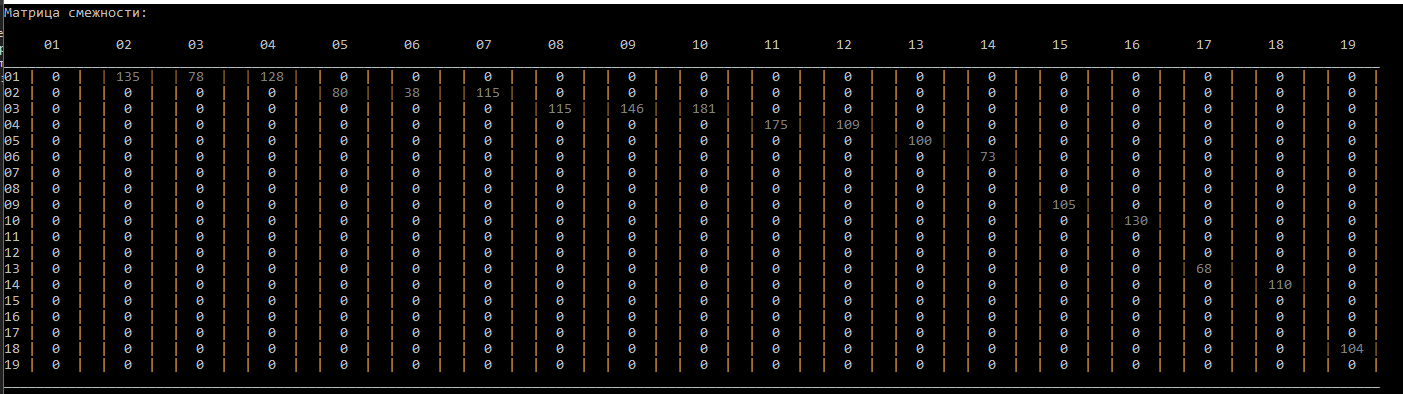


Рис.6.2.Результат

Також зробимо метод, який буде вказувати на можливі шляхи з вершин:

private static void GetVertex(Graph graph, Vertex vertex)

{

Console.Write(vertex.Number + ":");

foreach (var v in graph.GetVertexLists(vertex))

{

Console.Write(v.Number + " ");

}

Console.WriteLine();

}

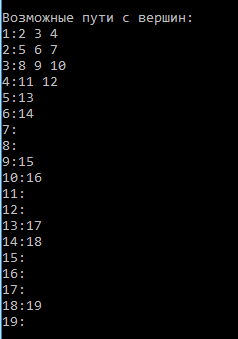


Рис.6.3.Результат

Для зображення можливих маршрутів створимо масив з назвами міст.

Також зробимо масив в яких буде зберігатися значення відстаней між містами.

static int[,] distance = new int[19, 19];

static string[] city = new string[19]

{

"Киев", "Житомир", "Новоград-Волынский", "Ровно", "Луцк", "Бердичев", "Винница", "Хмельницкий", "Тернополь", "Шепетовка", "Белая церковь", "Умань", "Черкассы", "Кременчуг", "Полтава", "Харьков", "Прилуки", "Сумы", "Миргород"

};

Утворимо матрицю та вузли для того, щоб створити шляхи:

public static void PossibleRoutes(int count, string prev, int weight)

{

string ways;

if (count == 0)

{

ways = city[count];

}

else

{

ways = prev + " - " + $"({weight})" + city[count];

}

Console.WriteLine(ways);

distance[0, 1] = 135;

distance[0, 10] = 78;

distance[0, 16] = 128;

distance[1, 2] = 80;

distance[1, 5] = 38;

distance[1, 9] = 115;

distance[2, 3] = 100;

distance[3, 4] = 68;

distance[5, 6] = 73;

distance[6, 7] = 110;

distance[7, 8] = 104;

distance[10, 11] = 146;

distance[10, 13] = 115;

distance[10, 14] = 181;

distance[11, 12] = 105;

distance[14, 15] = 130;

distance[16, 17] = 175;

distance[16, 18] = 109;

for (int i = 0; i < 19; i++)

{

if (check[i] == false && distance[count, i] != 0)

{

PossibleRoutes(i, ways, weight = distance[count, i]);

}

}

}

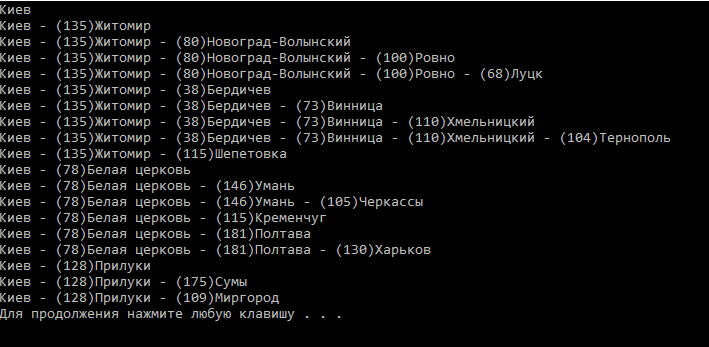


Рис.6.4.Результат

***Висновки:*** у ході виконання лабораторної роботи я дізнався про графи, про те, як і де вони використовуються, як з ними працювати, та як за їх допомогою розробити побудову шляхів на основі зв’язних вузлів. Я дізнався, як будувати матрицю сміжності та як її реалізувати у програмному коді на мові програмування C#.