НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«Київський Політехнічний Інститут імені Ігоря Сікорського»
Кафедра Технічної кібернетики

ЗВІТНІСТЬ ПО ПРАКТИЧНИМ РОБОТАМ З ДИСЦИПЛІНИ «АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ»

Виконав студент
групи ЗПІ-зп-02
Кононов М. А.

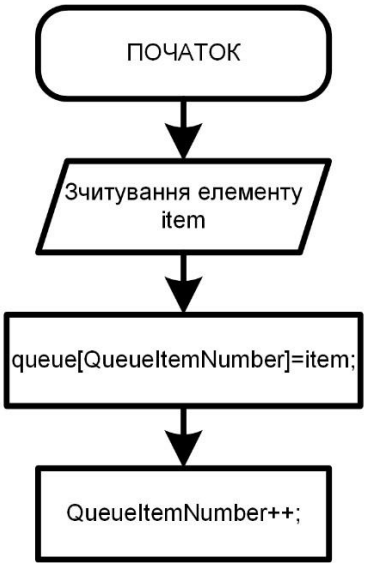
Мова програмування: C

Київ-2021

ПРАКТИЧНА РОБОТА №1. Абстрактні типи даних.

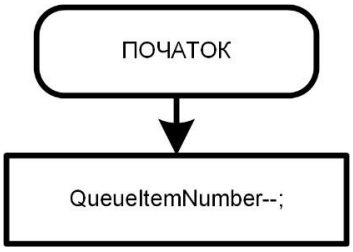
Постановка задачі: дослідження АТД «черга» та «двозв'язний лінійний список», реалізації за допомогою масивів та покажчиків на мові програмування С. Блок-схеми реалізацій «черги»:

Додавання елементу (масив)

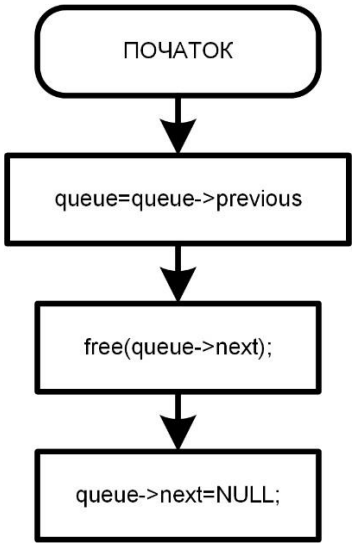


Додавання елементу (покажчик)

Видалення елементу (масив)



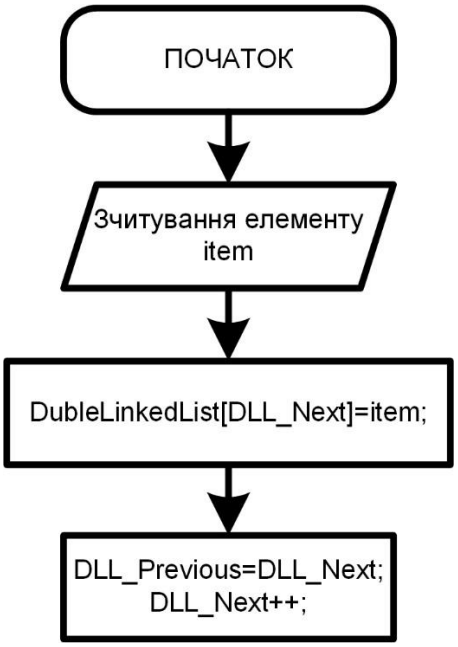
Видалення елементу (покажчики)



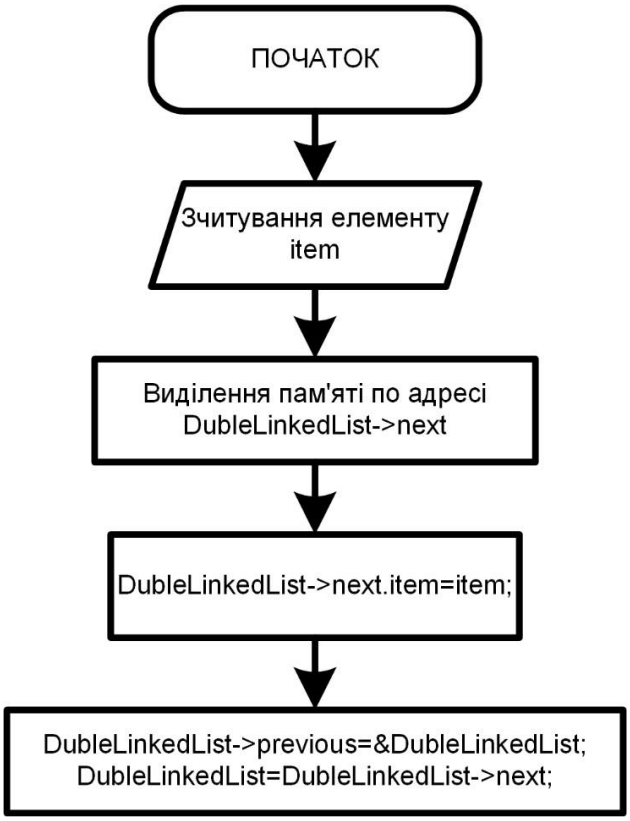
Складність усіх наведених алгоритмів складає O(1).

Блок-схеми реалізації АТД «двозв’язний лінійний список»:

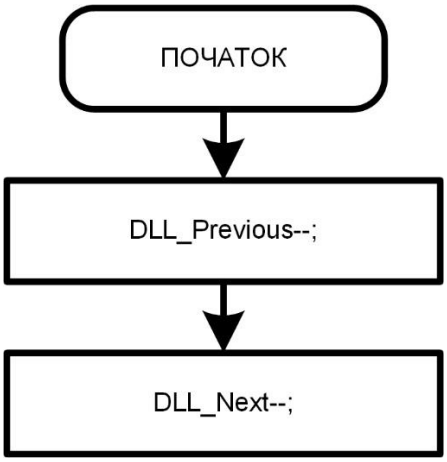
Додавання елементу (масив)



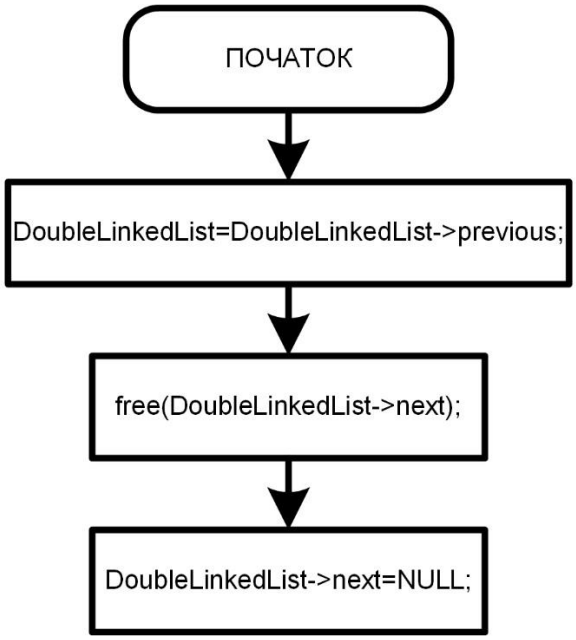
Додавання елементу (покажчик)

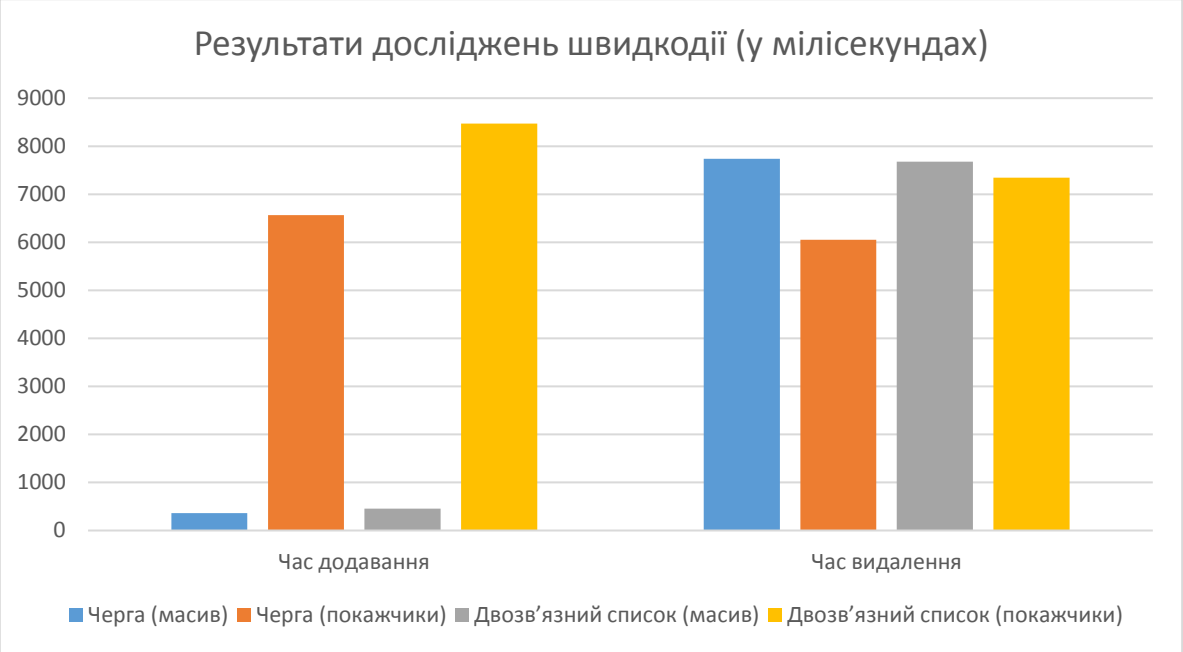


Видалення елементу (масив)



Видалення елементу (покажчики)



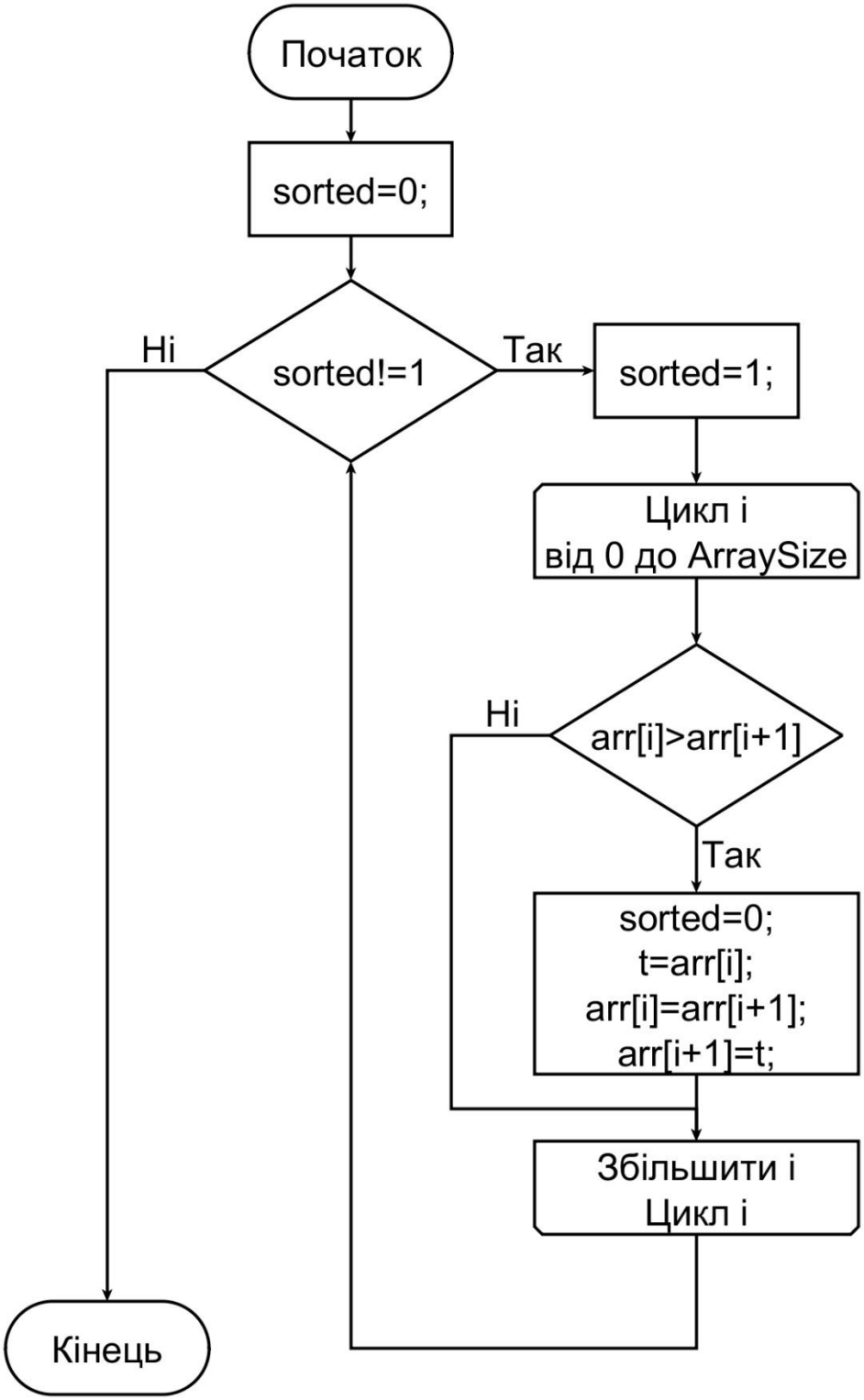


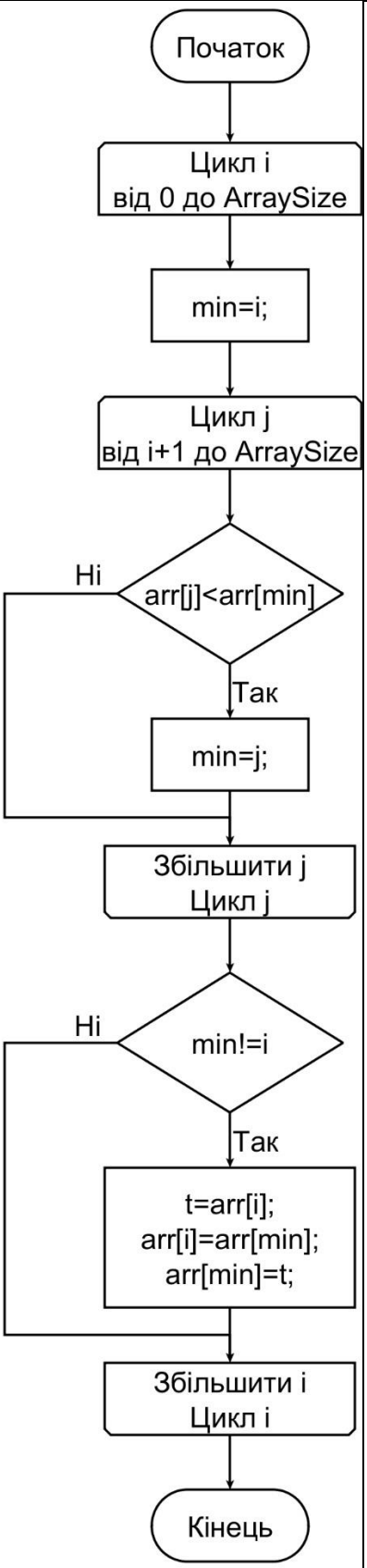
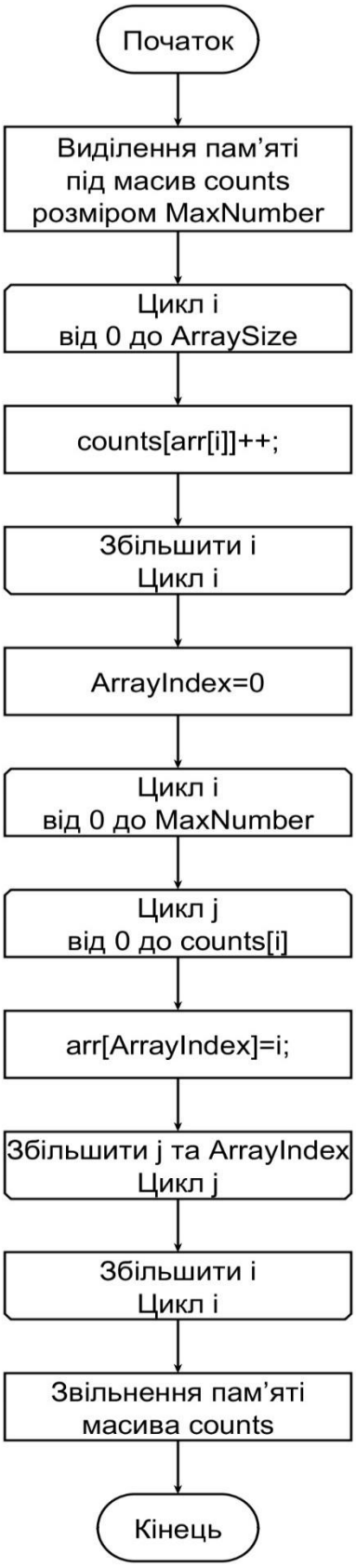
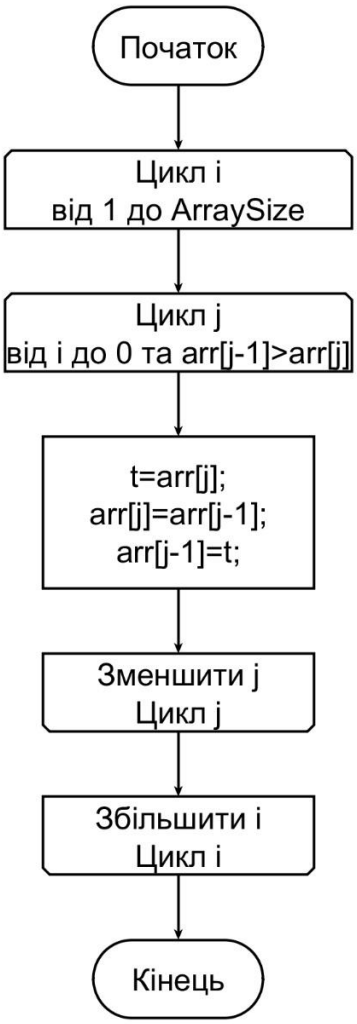
Висновок. Складність усіх наведених алгоритмів складає O(1). Тестові випробування виконувались над випадковими числами типу «*int*» у кількості 50 млн. Проаналізувавши швидкодію, ми дійшли висновку, що реалізації вибраних типів даних за допомогою масивів є найбільш швидкодіючими, потребують менше пам'яті та є простими з точки зору написання коду. Найбільш вибагливою до ресурсів процесора є операція видалення, оскільки реалізований нами алгоритм змінює розмір виділеної оперативної пам'яті. Вказані оцінки складності алгоритмів є аналітичними та, мабуть, повністю відповідають реальним результатам наведеного дослідження.

ПРАКТИЧНА РОБОТА №2. Алгоритми сортування. Методи сортування малих обсягів даних.

Постановка задачі: дослідження алгоритмів сортування методами бульбашки, вибору, простих вставок та підрахунку.

Схема алгоритму сортування бульбашкою

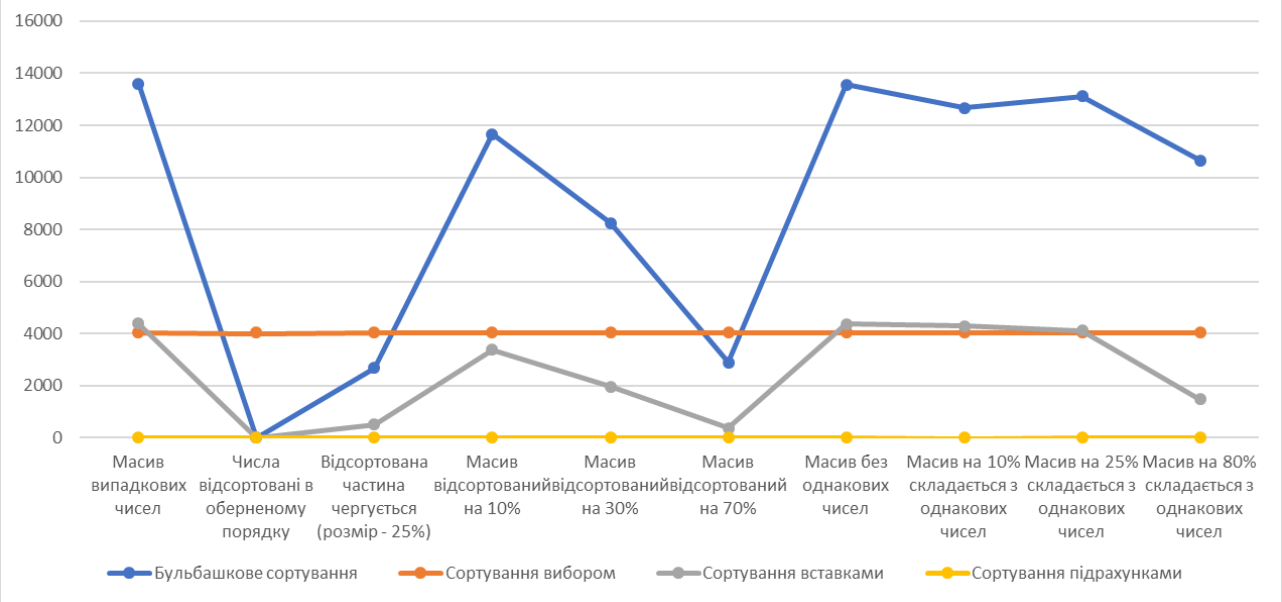


Сортування вибором	Сортування підрахунками	Сортування вставками
		

Порівняння складності реалізованих алгоритмів

Назва алгоритму	**Гірший випадок**	**Найкращий випадок**
Сортування бульбашкою	$O(n^2)$	O(n)
Сортування вибором	$O(n^2)$	$O(n^2)$
Сортування вставками	$O(n^2)$	O(n)
Сортування підрахунками	O(n+max-min)	O(n+max-min)

Дослідження швидкодії роботи (кількість елементів: 40000, час – мілісекундах)



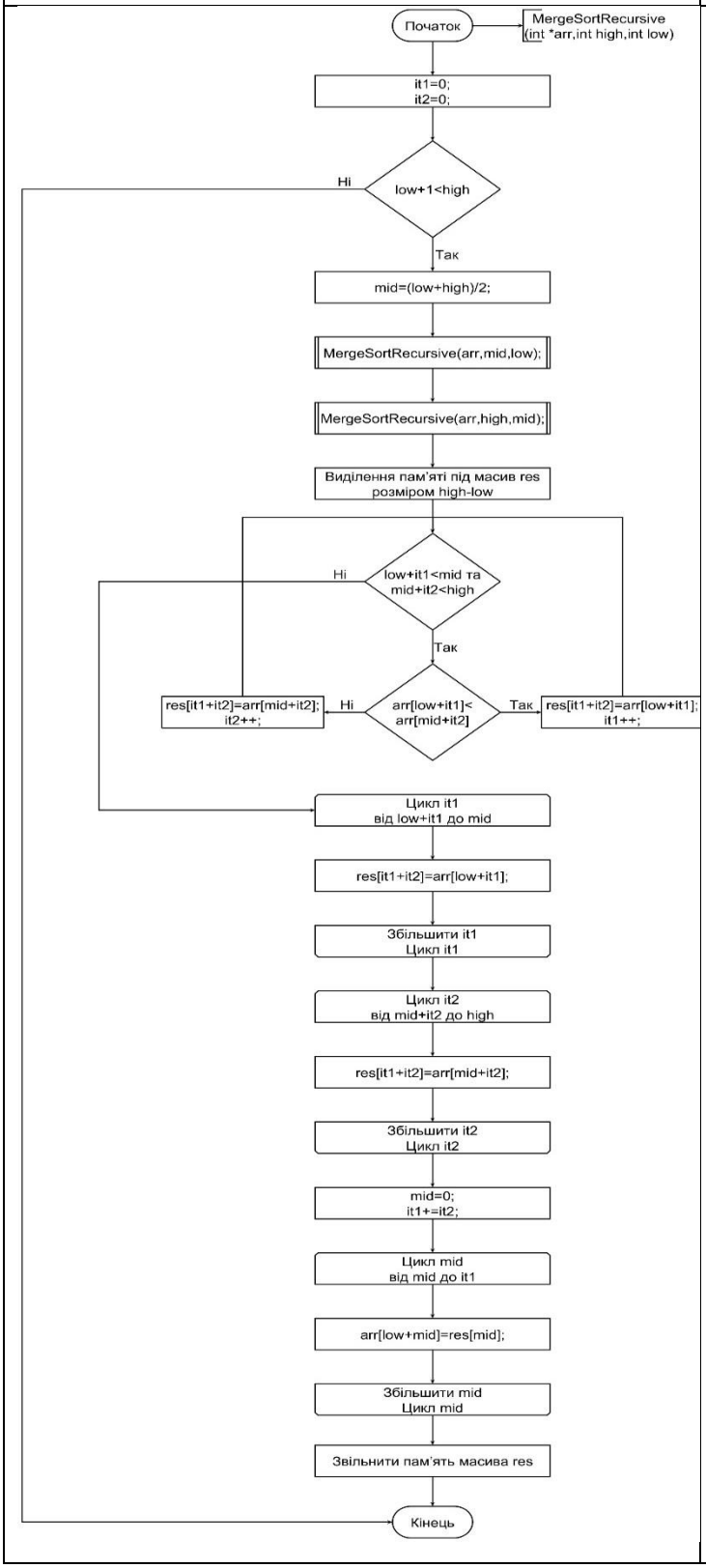
Висновок. У процесі виконання даної роботи нами були розроблені процедури, які генерують дані різного характеру: випадкові, в різній мірі відсортовані та з різним процентом вмісту однакових значень. Для тестування реалізованих рішень була розроблена процедура, яка перевіряє вихідний масив на наявність всіх чисел із джерела та коректність порядку їх розміщення.

Було встановлено, що способи сортування підрахунками та вибором є найменш залежними від вхідних даних. Бульбашковий алгоритм є найгіршим у майже всіх випадках, але його доцільно використовувати, якщо дані впорядковані в оберненому порядку. Сортування підрахунками є найкращим за швидкодією, але використання надто великих чисел, які перевищують вільний об’єм оперативної пам’яті, може призвести до переривання роботи програми.

ПРАКТИЧНА РОБОТА №3. Алгоритми сортування. Методи сортування великих обсягів даних.

Постановка задачі: дослідження алгоритмів швидкого, плавного, пірамідального сортування, а також сортування злиттям.

Схема алгоритму сортування злиттям



Швидке сортування (рекурсія)

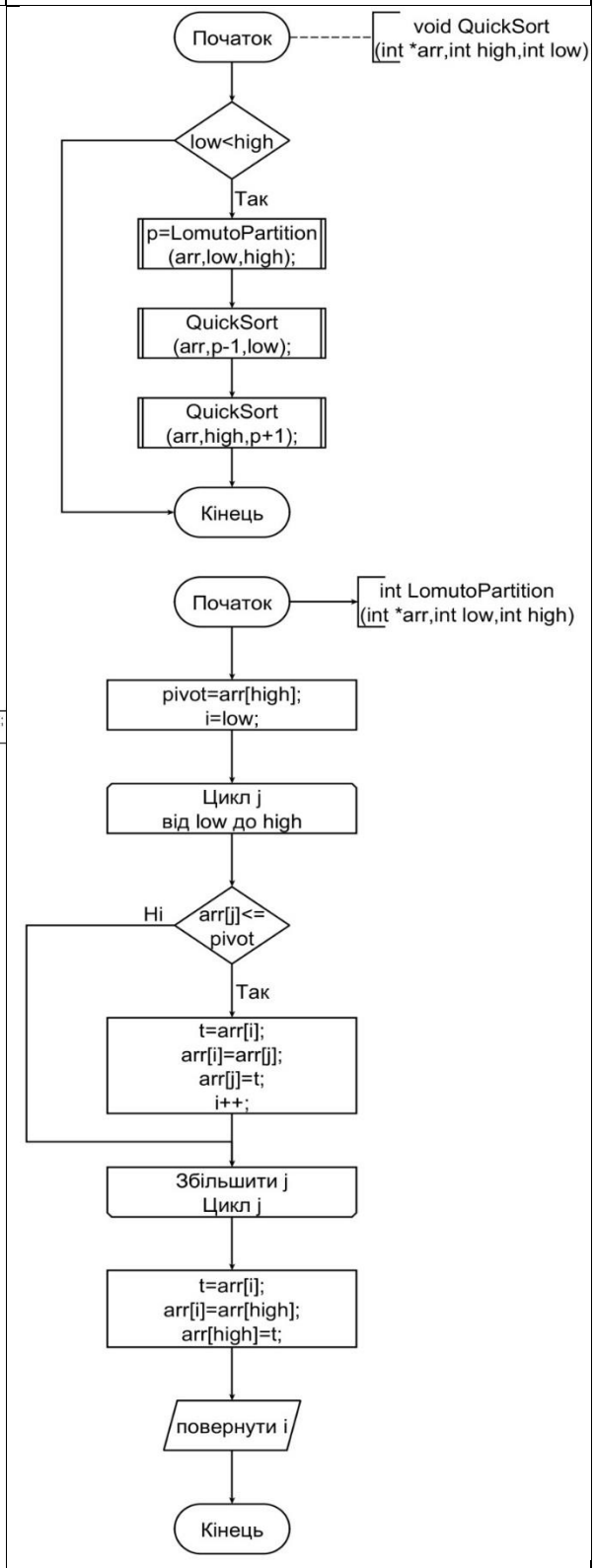
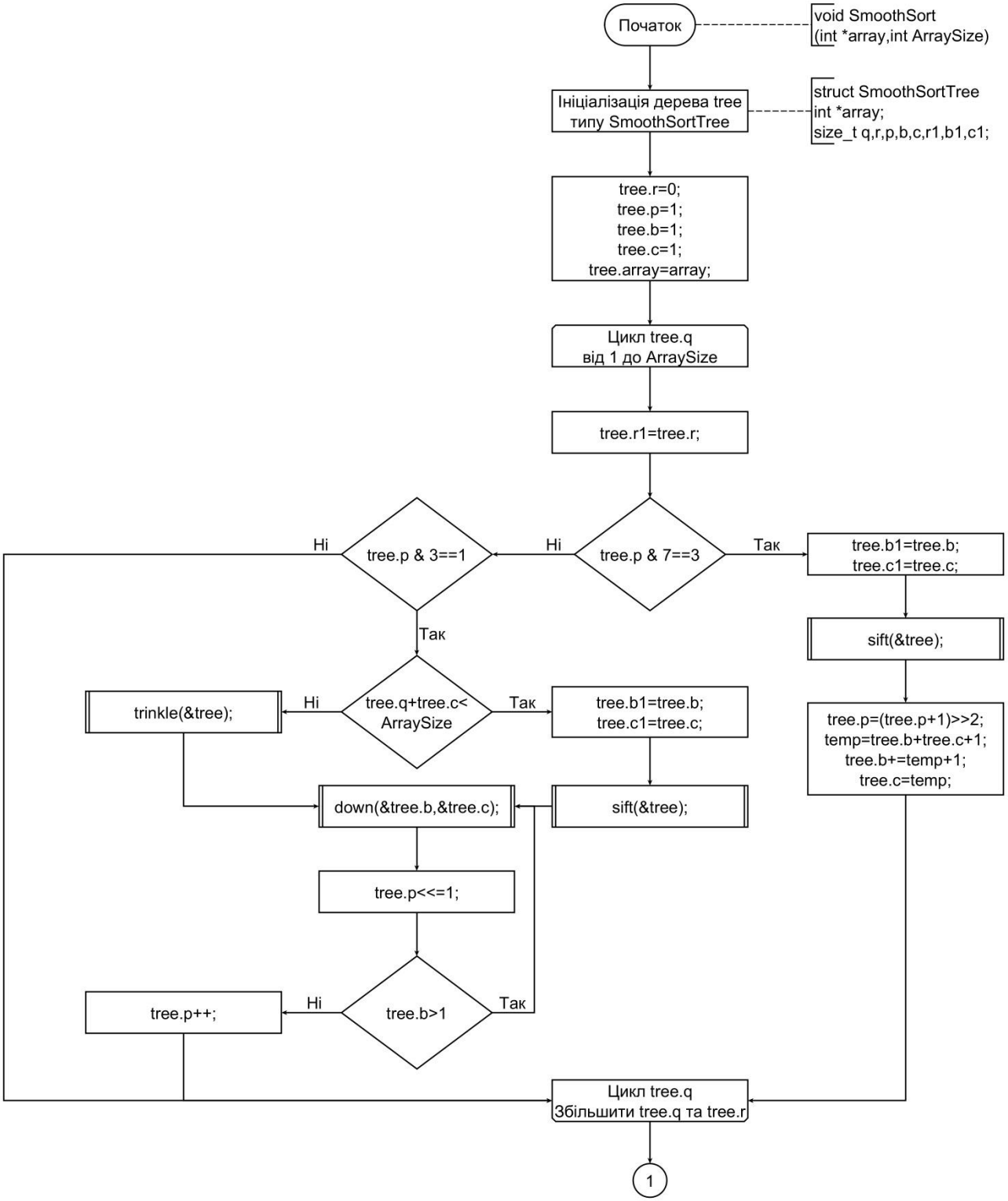
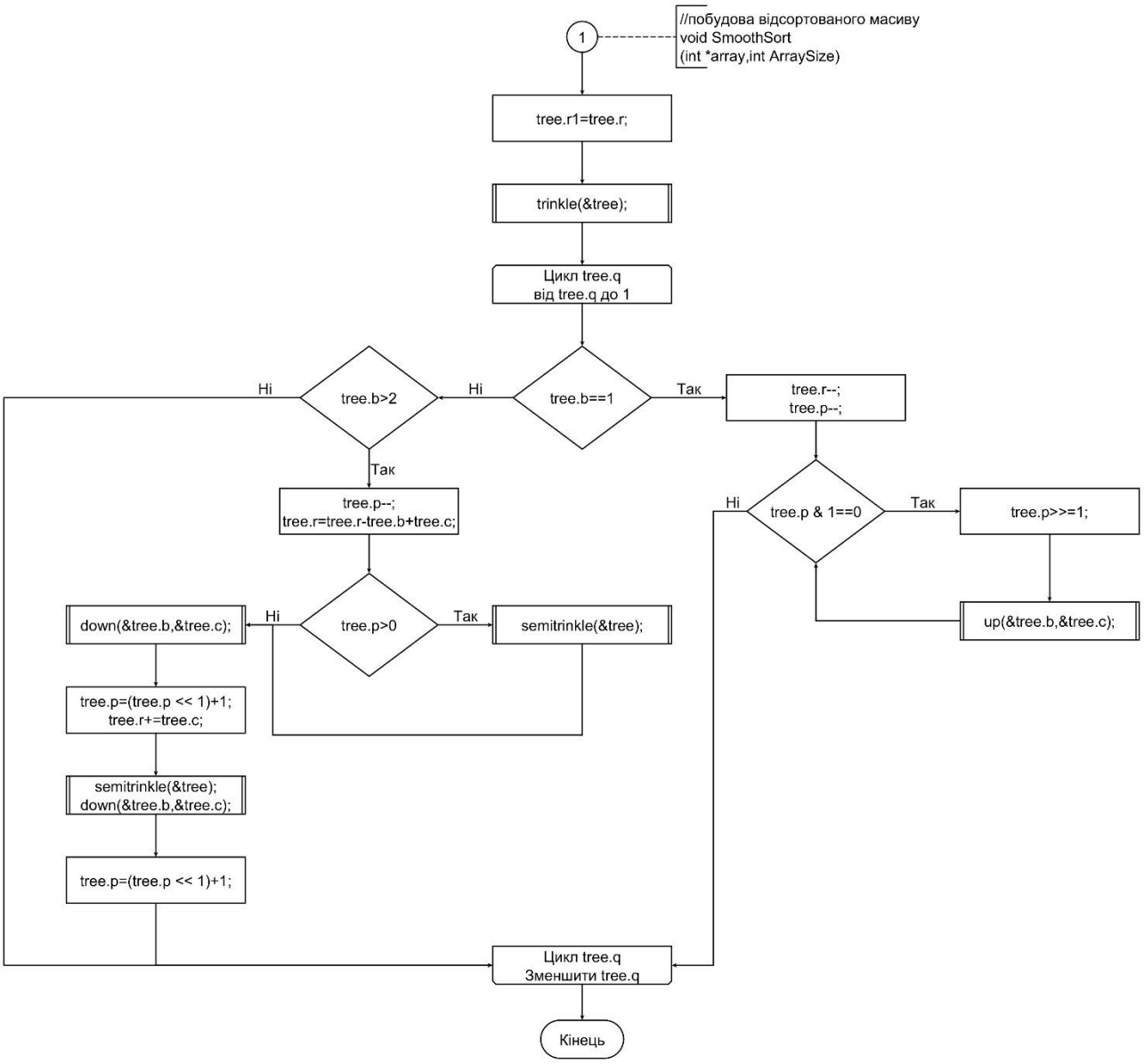
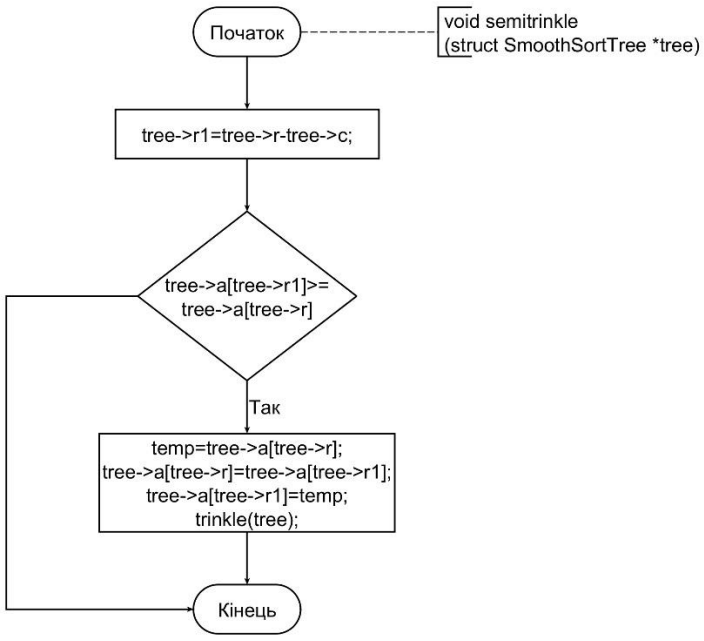


Схема алгоритму плавного сортування

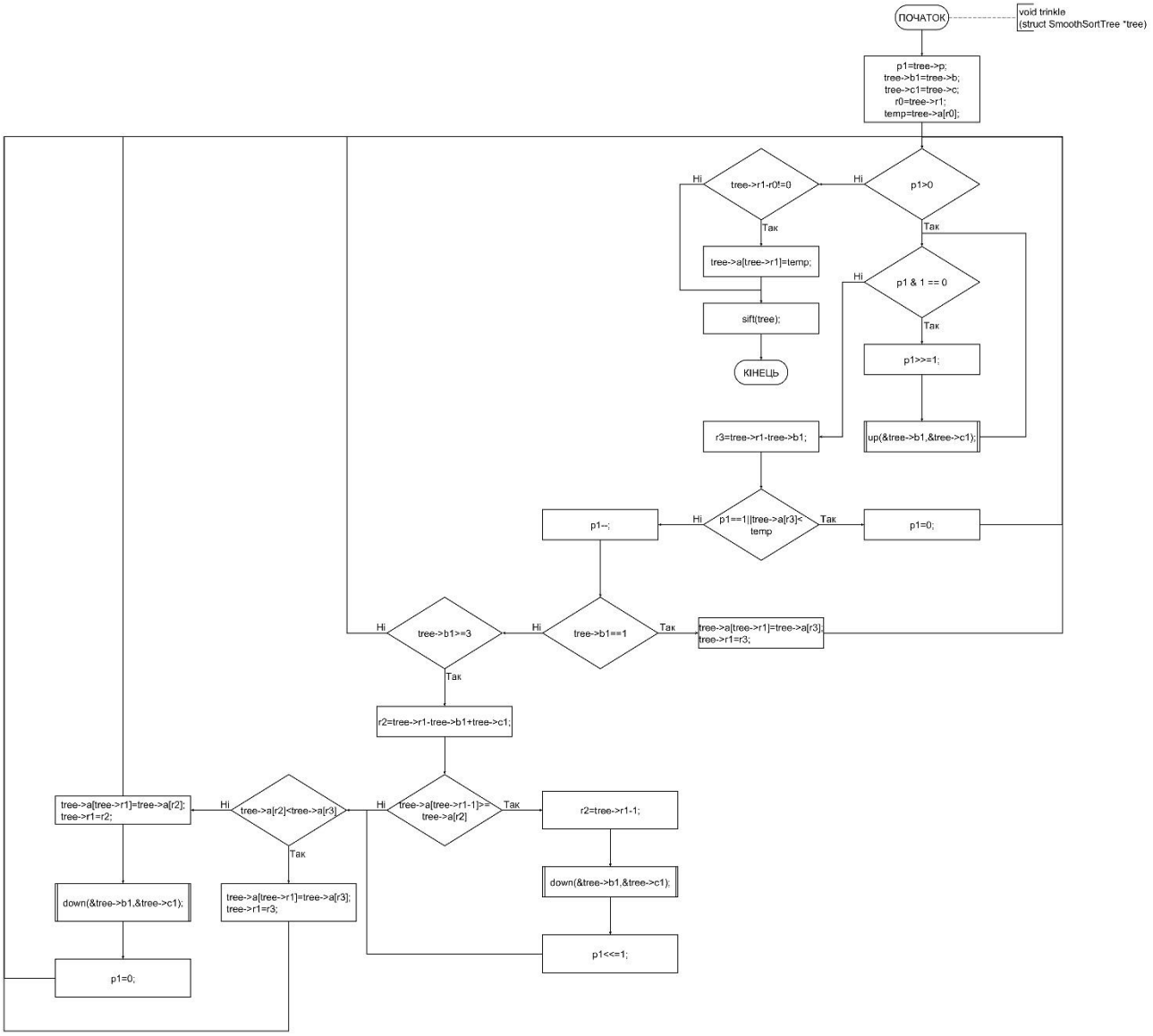




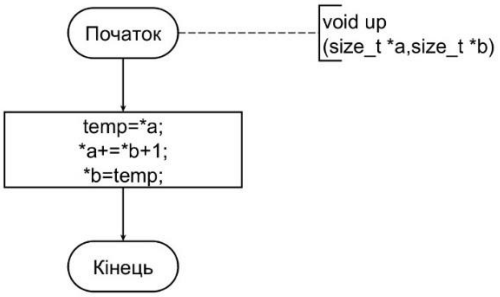
Процедура semitrinkle

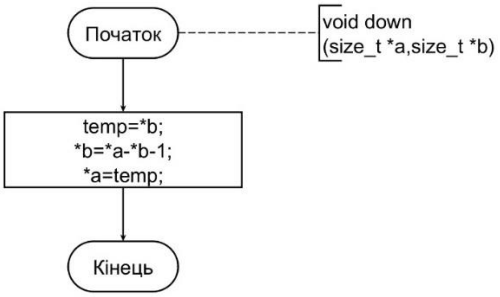


Процедура trinkle

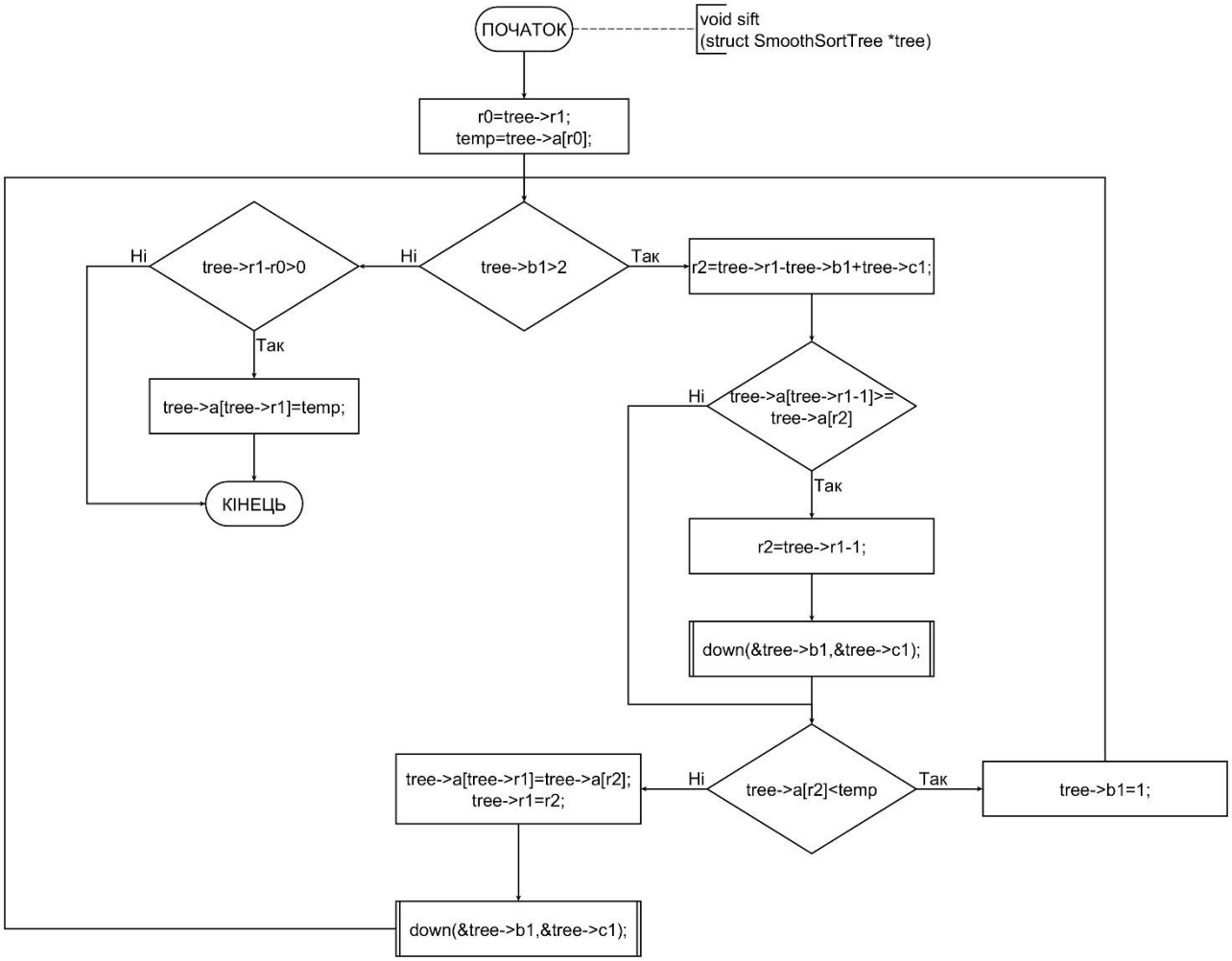


Процедури Up та Down





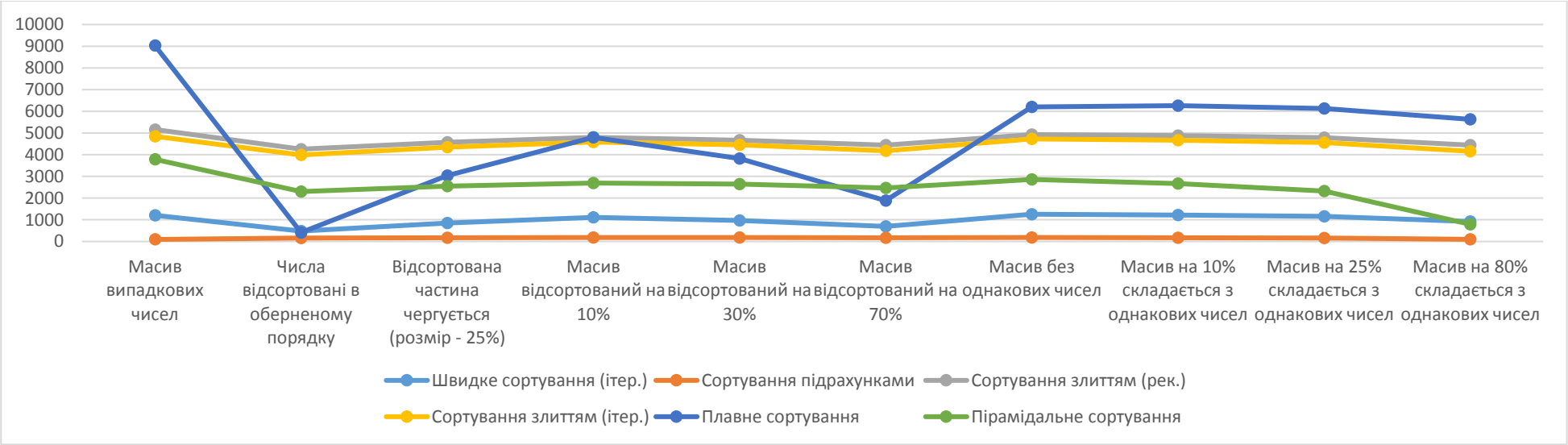
Процедура sift

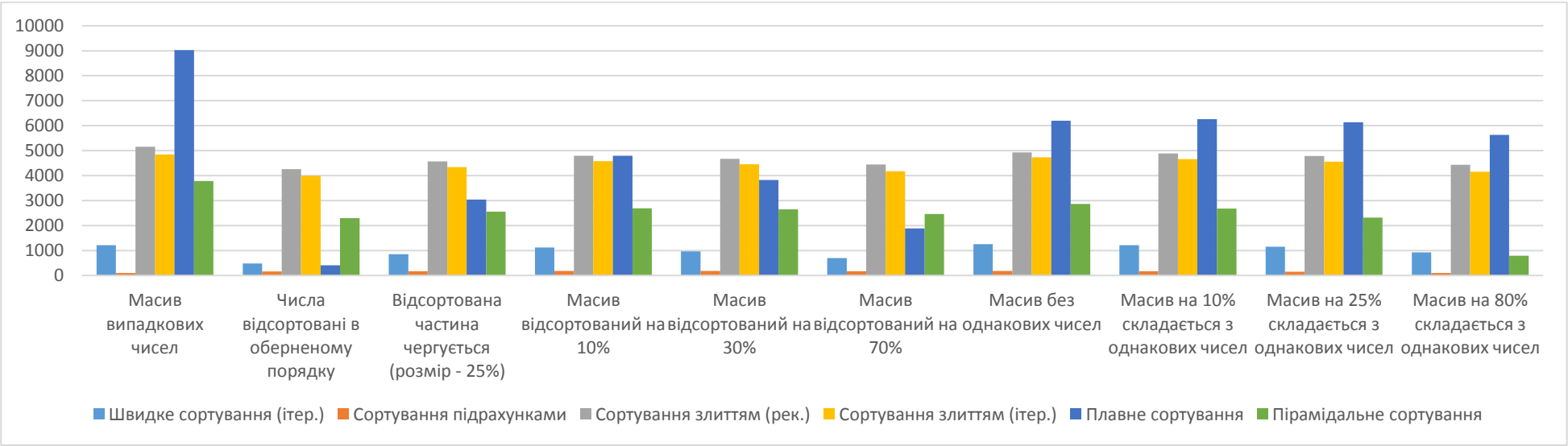


Складність реалізованих алгоритмів

Назва алгоритму	Найгірший випадок	Середній випадок	Найкращий випадок
Сортування злиттям	n*log (n)	n*log (n)	n*log (n)
Плавне сортування	n*log (n)	n*log (n)	N
Швидке сортування	$O(n^2)$	O(n log n)	O(n log n)
Пірамідальне сортування	O(n log n)	O(n log n)	O(n log n)

Дослідження швидкодії роботи (кількість елементів: 5000000, час – мілісекундах)



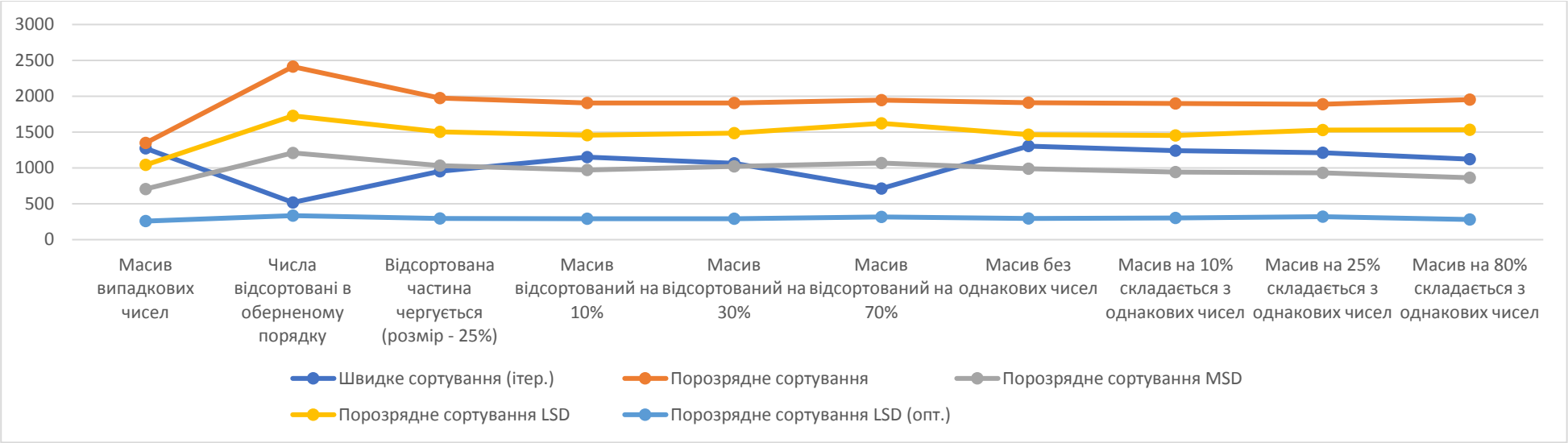


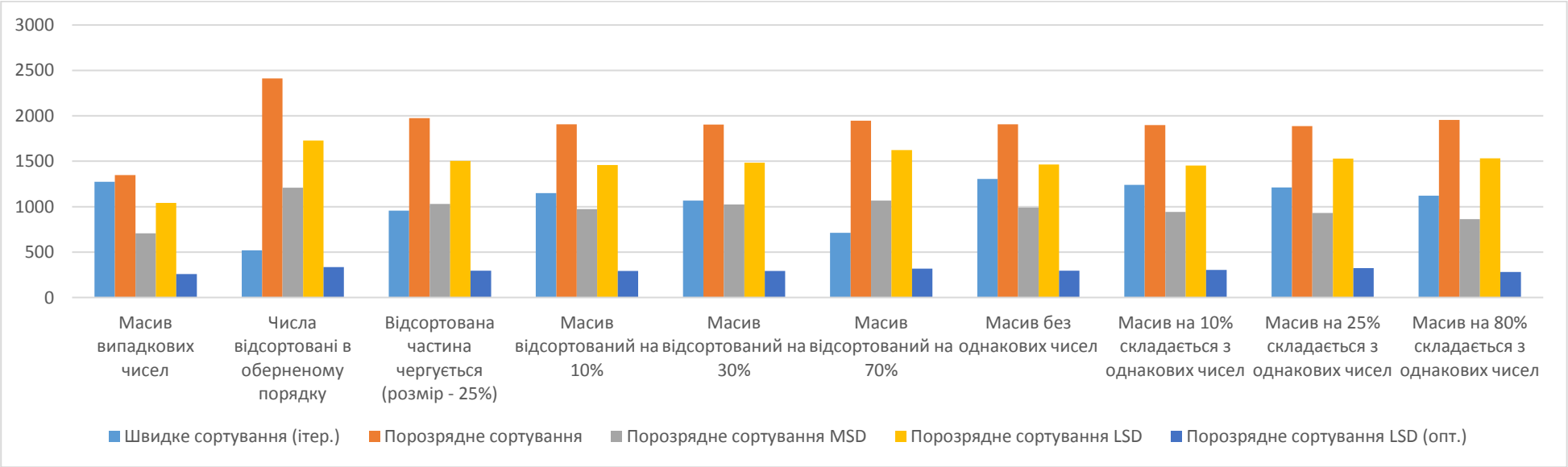
Висновок. У процесі дослідження було встановлено, що швидкодія ітеративного та рекурсивного алгоритмів сортування злиттям знаходяться майже на одному рівні та їх ефективність майже не залежить від характеру даних. Плавне сортування сильно залежить від вхідних чисел: масив, відсортований в оберненому порядку оброблюється найбільш ефективно, а випадковий – найменш ефективно. Оскільки класична рекурсивна реалізація швидкого сортування є дуже повільною на великому об’ємі вхідних даних та її використання може призвести до переповнення стеку викликів операційної системи і некоректного завершення програми при роботі з числами, розміщеними в оберненому порядку, ми дослідили оптимізовану ітеративну версію цього алгоритму. Нами було встановлено, що воно не є найшвидшим методом, і у більшості випадків не залежить від типу чисел у масиві. Сортування підрахунками, описане в попередній роботі, виявилось найшвидшим. Можна зробити висновок, що пірамідальний алгоритм впорядкування слабо залежить від вхідних даних та займає середню позицію в ієрархії ефективності досліджених алгоритмів.

ПРАКТИЧНА РОБОТА №4. Алгоритми сортування. Методи сортування даних з довгими ключами.

Постановка задачі: дослідження стандартного алгоритму порозрядного сортування, алгоритму LSD та алгоритму MSD, їх порівняння із швидким та порозрядним сортуванням.

Дослідження швидкодії роботи (кількість елементів: 5000000, час – мілісекундах)





Складність алгоритмів

Назва алгоритму	**Складність**
Порозрядне сортування	O(n*k)
Порозрядне сортування LSD (оптимізована версія)	O(n)

Висновок. У процесі дослідження нами було реалізовано як класичні варіанти порозрядного сортування, так і варіант авторської оптимізації, який з'явився на початку 2020 року на сайті habr.com. Було встановлено, що оптимізована версія алгоритму LSD є найбільш ефективною та майже повністю не залежить від характеру вхідних даних. На відміну від усіх інших алгоритмів вона не здійснює порівняння та обмін елементів масиву. Суть даного алгоритму полягає в наступному:

1. Для кожного блоку (по 8 двійкових розрядів – оптимальна величина) вхідного масиву методом підрахунку обчислюється його положення в новому масиві;
2. Послідовно для кожного блоку (від молодших розрядів до старших) виконується переміщення даних на обчислену раніше відстань.
3. Для масиву зміщень використовується вирівнювання пам'яті, а завдяки його невеликому об'єму, він може легко вміститися у кеші процесора;
4. Масив зміщень заповнюється лише один раз для всіх розрядів;

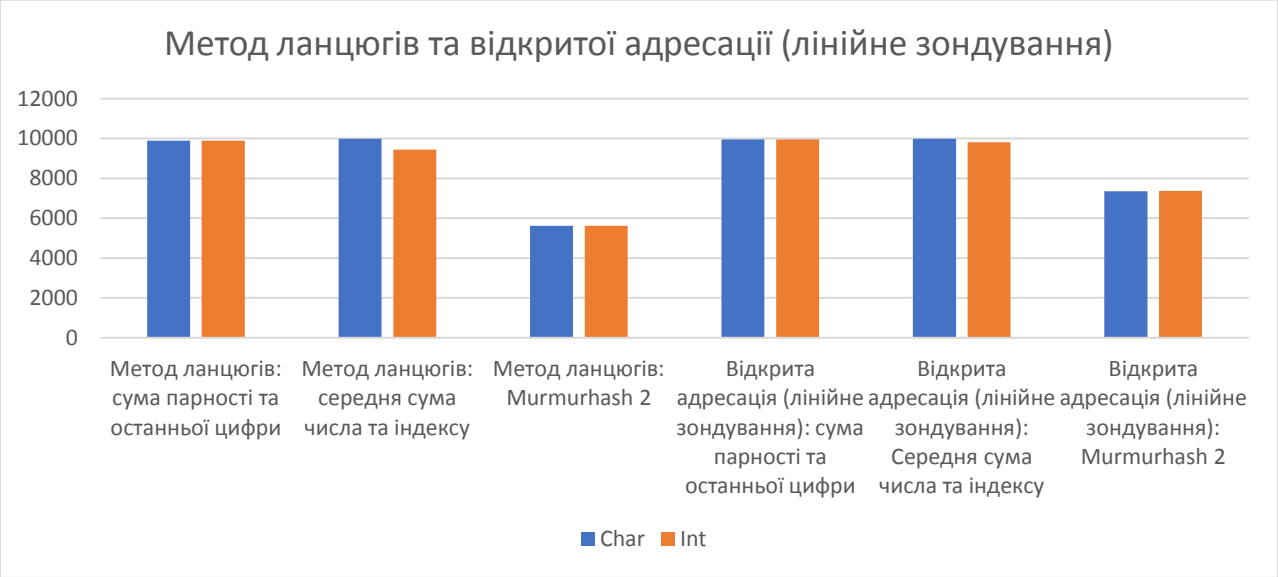
Класична версія порозрядного сортування виявилась найповільнішою, не набагато кращим є й стандартний алгоритм LSD. Їх ефективність не залежать від характеру вхідних даних. Швидке та порозрядне сортування MSD знаходяться майже на однаковому рівні. Ми дійшли висновку, що оптимізована авторська версія алгоритму LSD є найбільш ефективною з усіх досліджених нами алгоритмів.

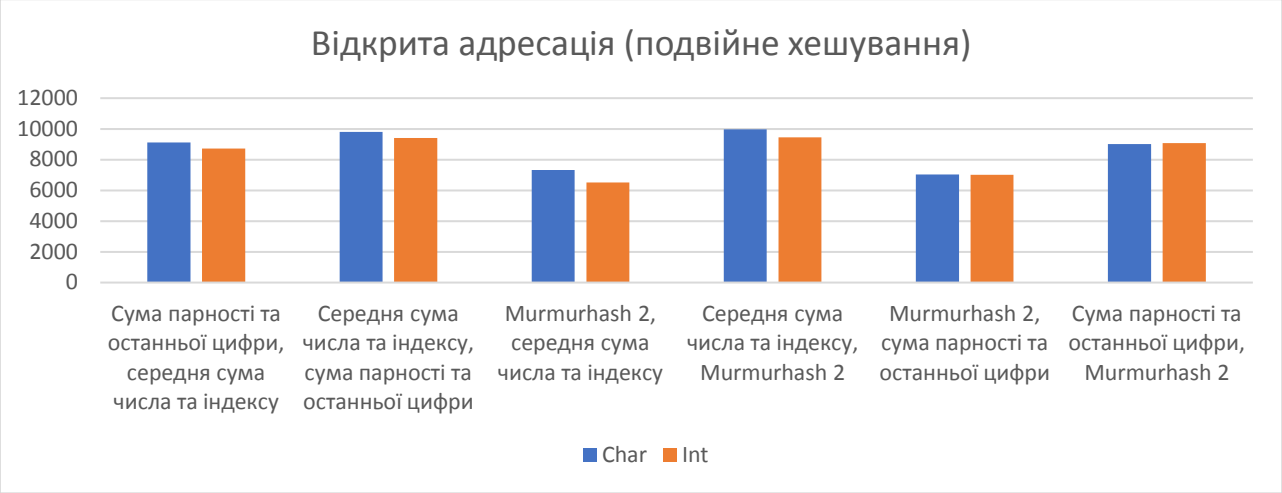
ПРАКТИЧНІ РОБОТИ №5-6. Пошук у масиві. Закрите хешування. Відкрите хешування.

1. Постановка задачі: дослідження трьох способів вирішення колізій при хешуванні: метод ланцюгів, відкрита адресація (лінійне зондування), відкрита адресація (подвійне хешування) з використанням трьох хеш-функцій: сума парності та останньої цифри, середня сума числа та його індексу в масиві, авторська функція Murmurhash 2. Усі випробування здійснювались на масивах з даними типів *int* та *char* (у хеш-таблицю додавались масиви випадкових даних однакового розміру). Результати випробувань з наступними параметрами:

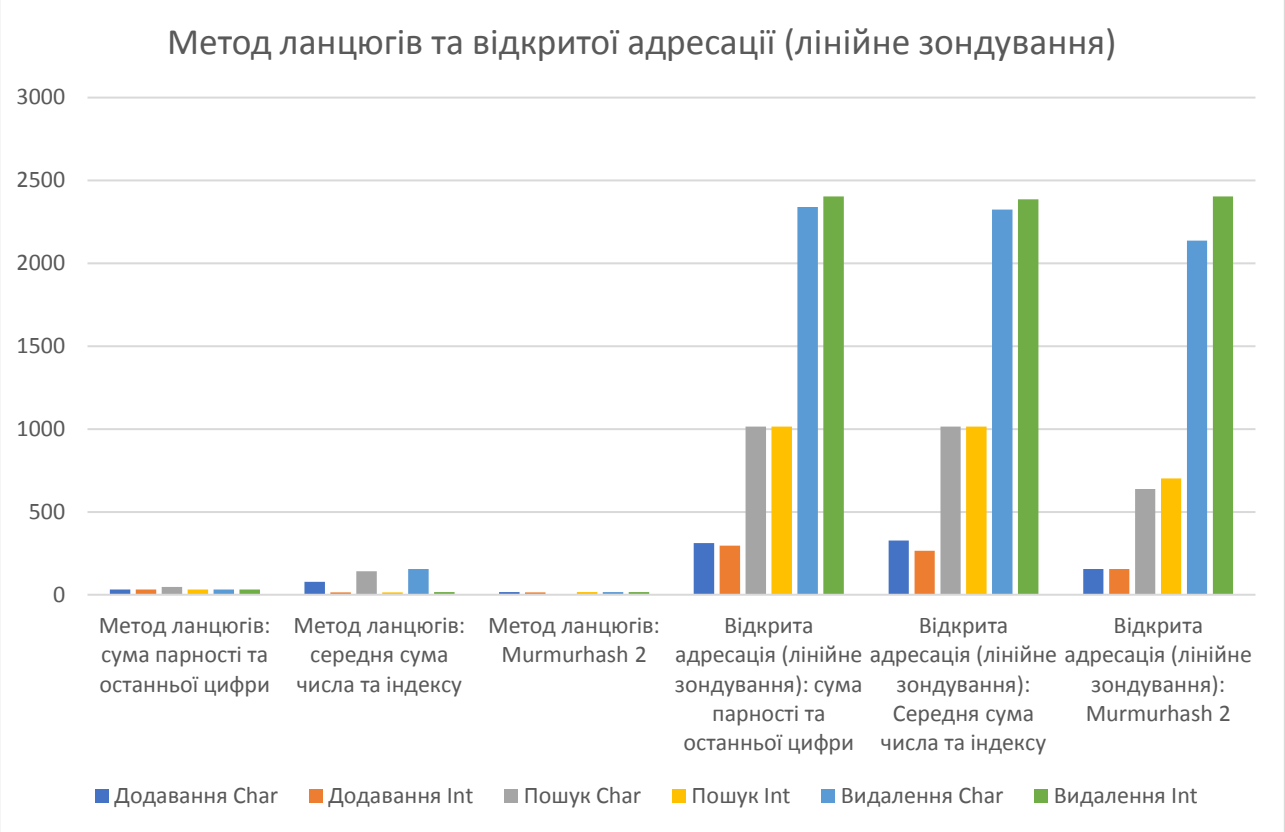
Розмір одного елементу таблиці	100 чисел
Кількість елементів, над якими здійснювались операції додавання, пошуку та видалення	10000
Розмір хеш-таблиці (максимальна кількість елементів у ній)	10000

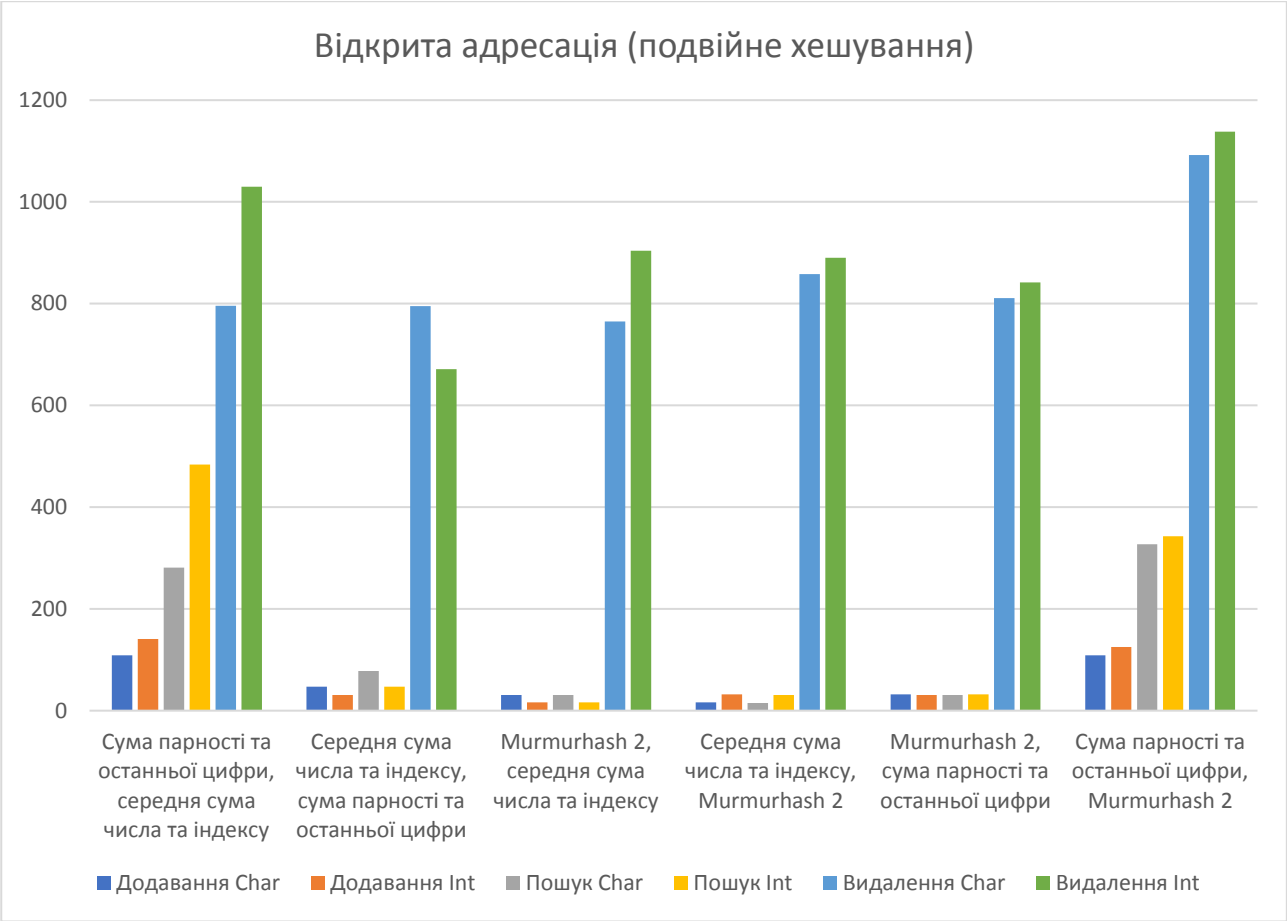
Порівняння кількості колізій



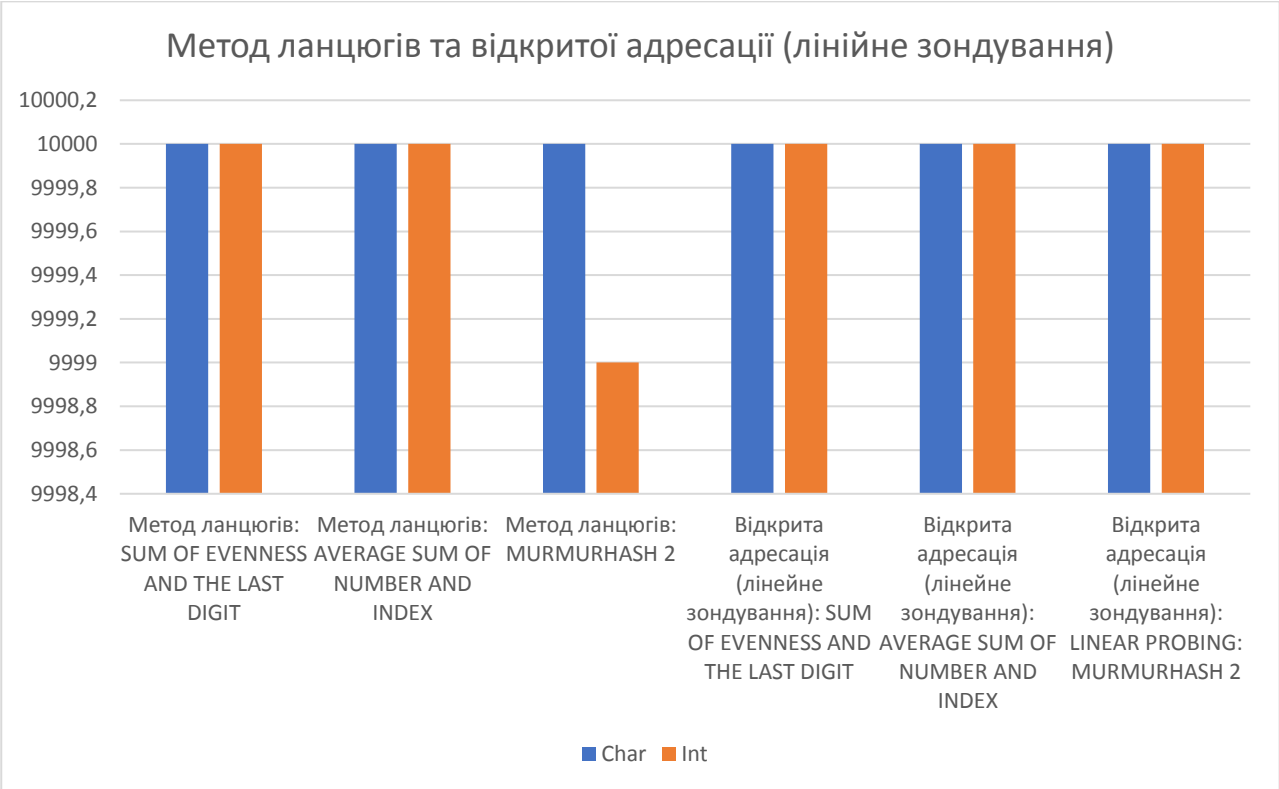


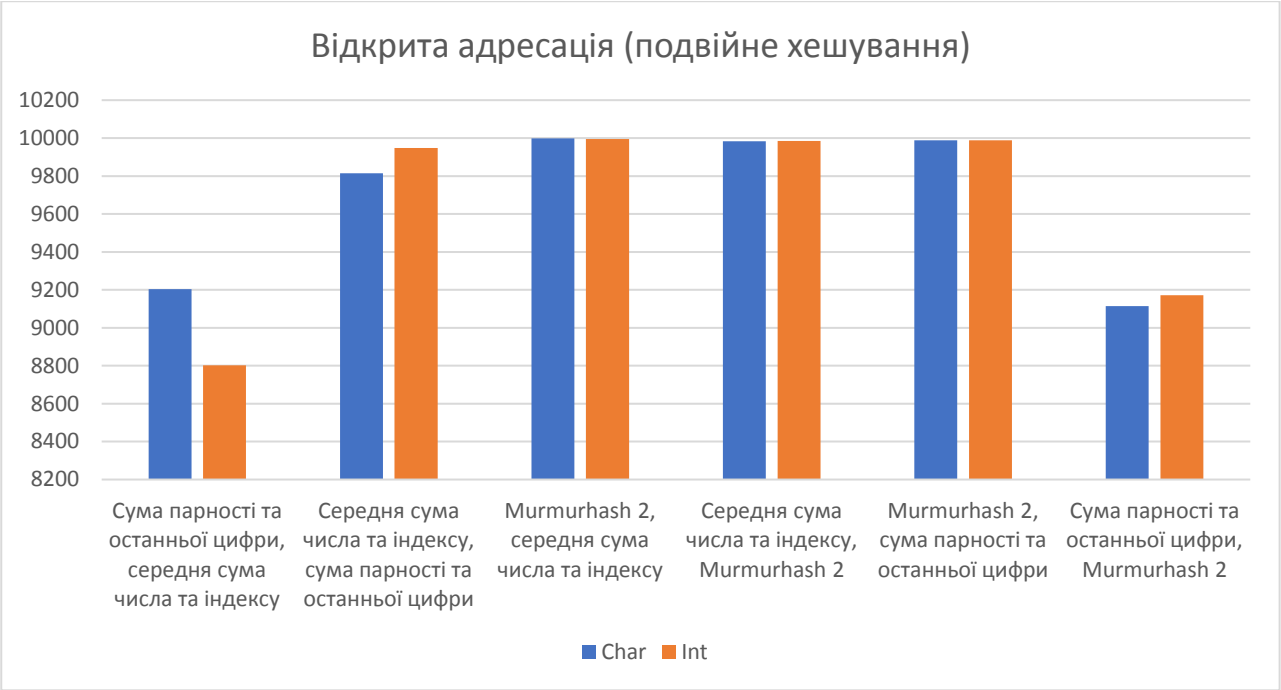
Порівняння часових витрат на виконання всіх операцій над усіма елементами (у мілісекундах)





Порівняння кількості доданих елементів (операції вдалого пошуку)





2. Висновок. Проаналізувавши три методи вирішення колізій з виконанням трьох хеш-функцій над масивами двох типів даних *int* та *char*, ми дійшли висновку, що найбільш стійкою до колізій є хеш-функція Murmurhash 2, розроблена Остином Епплбі. На другому місті – сума парності та останньої цифри, найбільшу кількість колізій викликає функція обчислення середньої суми числа та його індексу в масиві. Для збільшення кількості доданих елементів після виконання кожної хеш-функції циклічно виконувалась операція ділення на 10 доки значення хешу не стало меншим за максимальний розмір хеш-таблиці для збільшення ймовірності додавання елементу саме за обчисленою адресою. Ефективність (швидкодія виконання всіх операцій та стійкість до колізій) розроблених нами рішень залежить від заданих параметрів: розміру масивів, їх кількості, максимального розміру хеш-таблиці. Найбільш вибагливою до часу та ресурсів процесора є операція видалення при використанні методів відкритої адресації (лінійного зондування та подвійного хешування), оскільки після неї виконується зміщення багатьох доданих елементів на найкращу для них позицію.

При використанні методів ланцюгів та відкритої адресації засобом лінійного зондування хеш-таблиця може бути заповнена на 100%, тому реструктуризацію потрібно проводити лише в тому випадку, якщо кількість

елементів, які ми бажаємо додати, перевищує максимальний заданий розмір хеш-таблиці. Метод відкритої адресації засобом подвійного хешування не дає можливість відслідкувати коли потрібно збільшити розмір, оскільки можливість додавання елементу повністю залежить від значень обох хеш-функцій. Для більш точної оцінки швидкодії роботи всіх алгоритмів ми не реалізували можливість зупинки циклічного виконання операції додавання у випадку відсутності місця для нових елементів та сигналізацію про необхідність реструктурувати дані. Якщо елемент не може бути доданий з причини перевищення значення хеш-функції заданого розміру таблиці або повного заповнення таблиці при використанні методів відкритої адресації, алгоритм просто збільшить лічильник недоданих елементів.

ПРАКТИЧНА РОБОТА №7. Дерева пошуку.

1. Постановка задачі: дослідження бінарного та червоно-чорного дерев з використанням різних типів вхідних даних (випадкові, з різним ступенем відсортованості та різною кількістю однакових значень), порівняння швидкодії реалізації основних операцій з абстрактними типами даних «черга» та «двозв’язний лінійний список».

Схема алгоритму додавання елементу в бінарне дерево

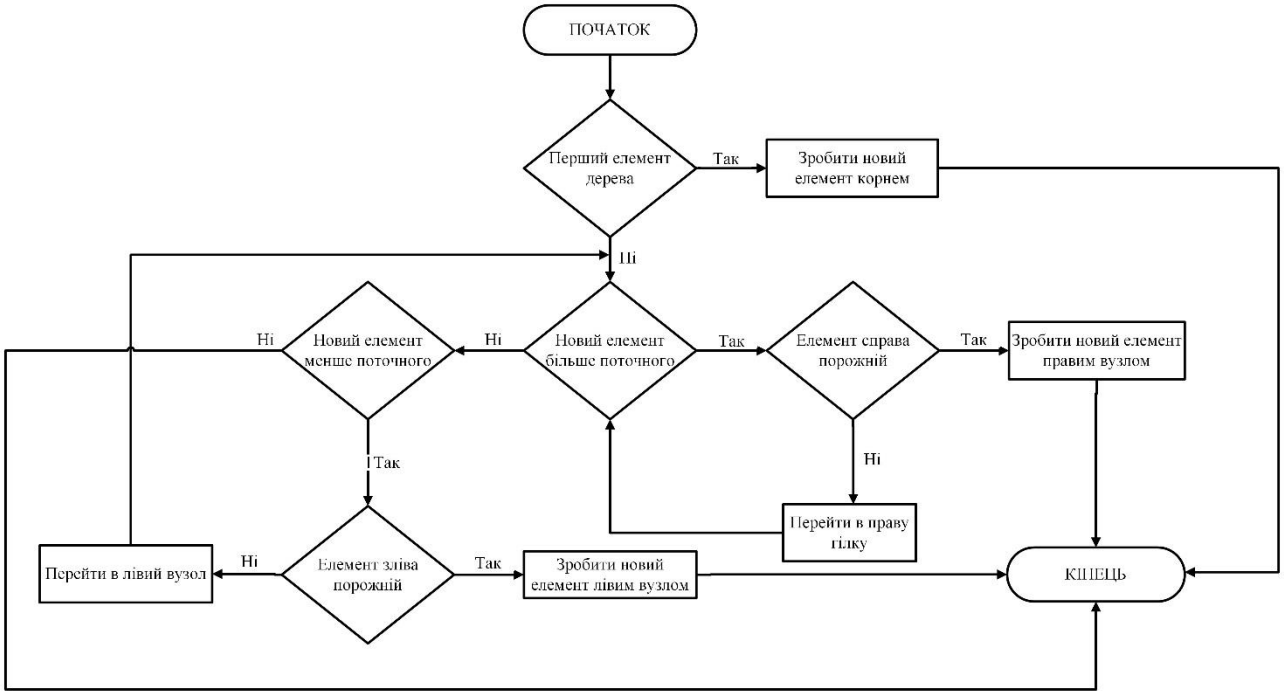
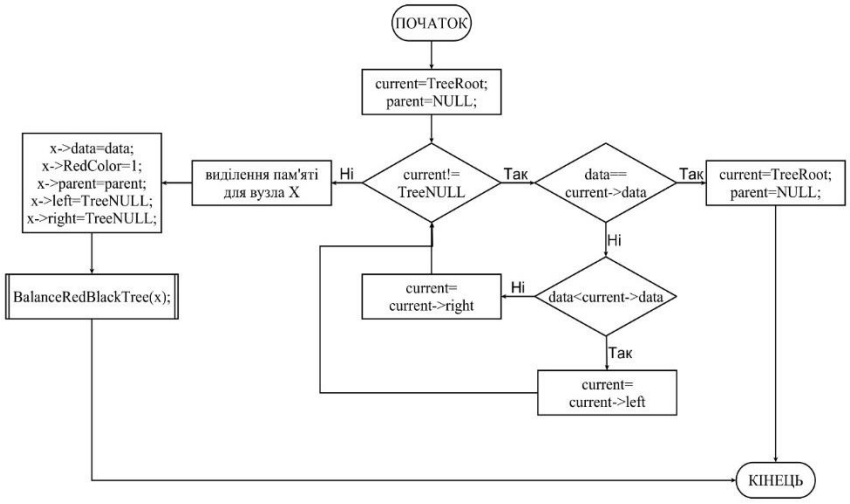
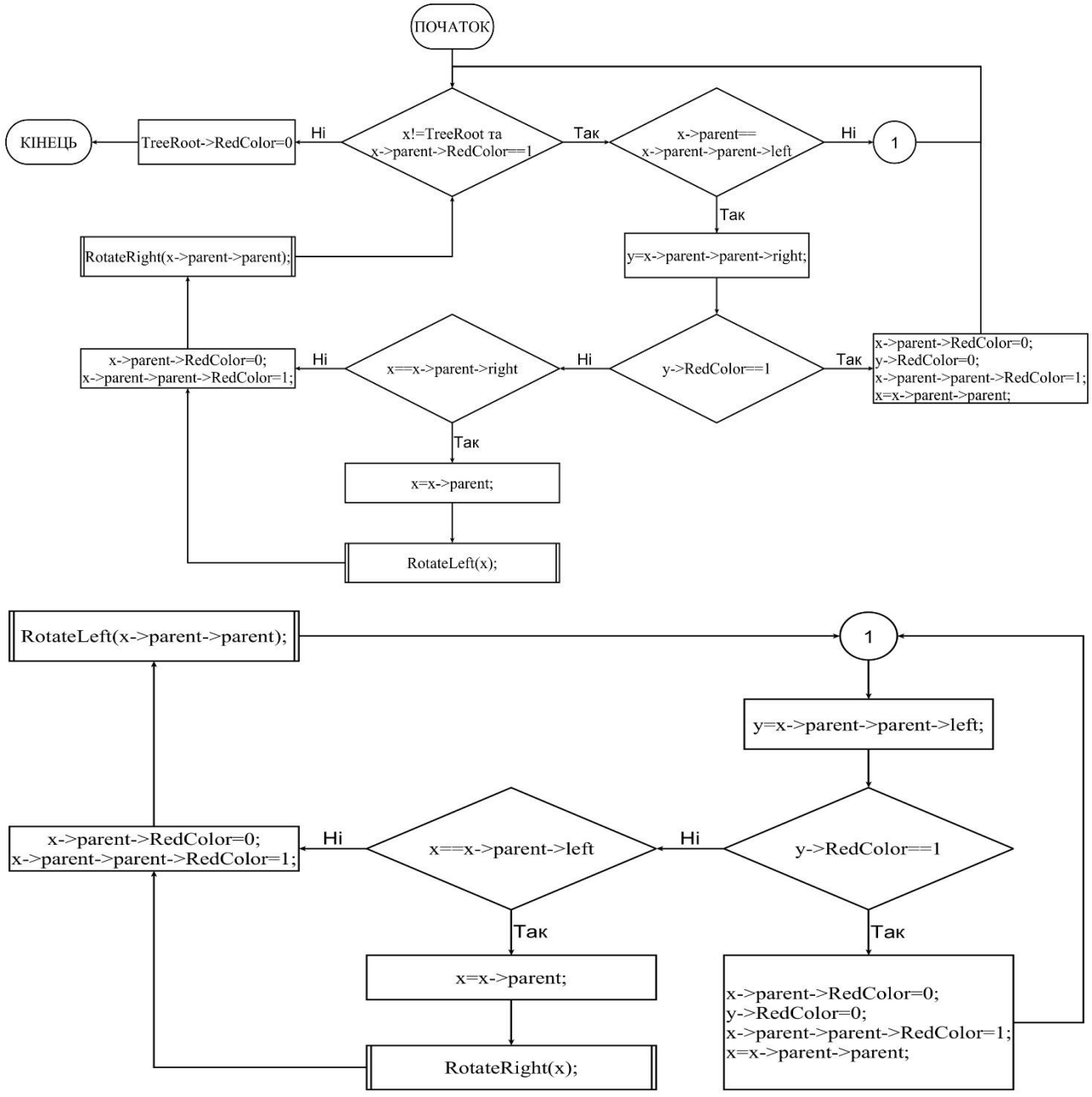


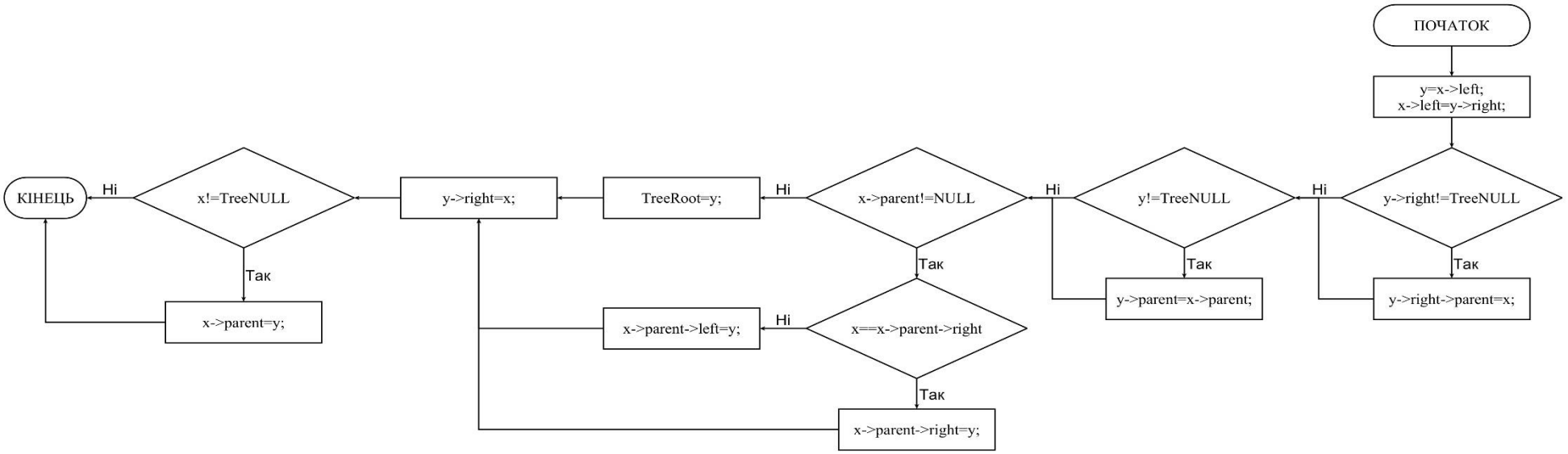
Схема алгоритму додавання елементу в червоно-чорне дерево



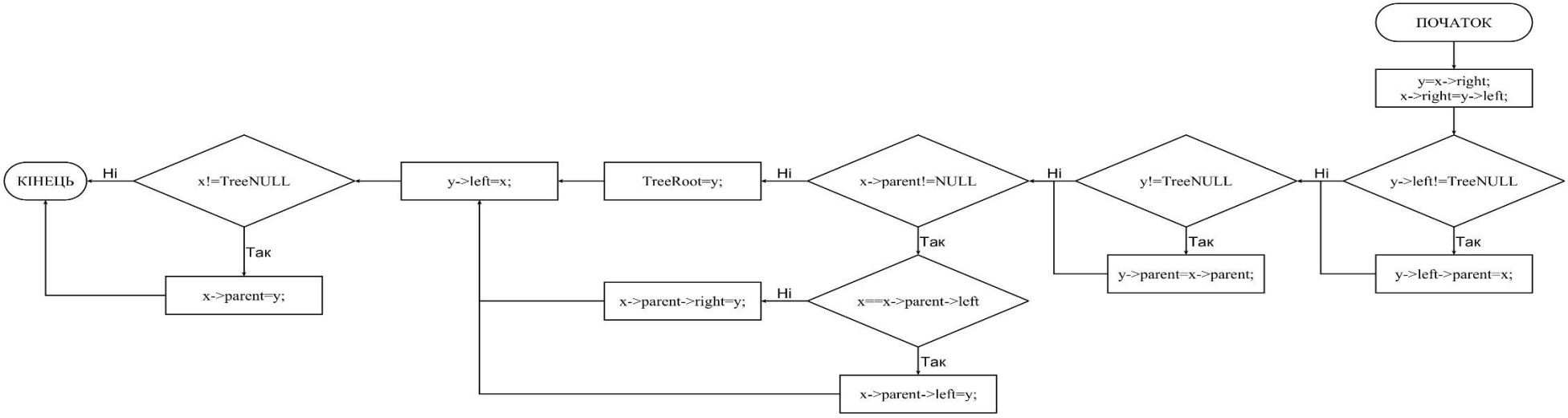
Балансування червоно-чорного дерева



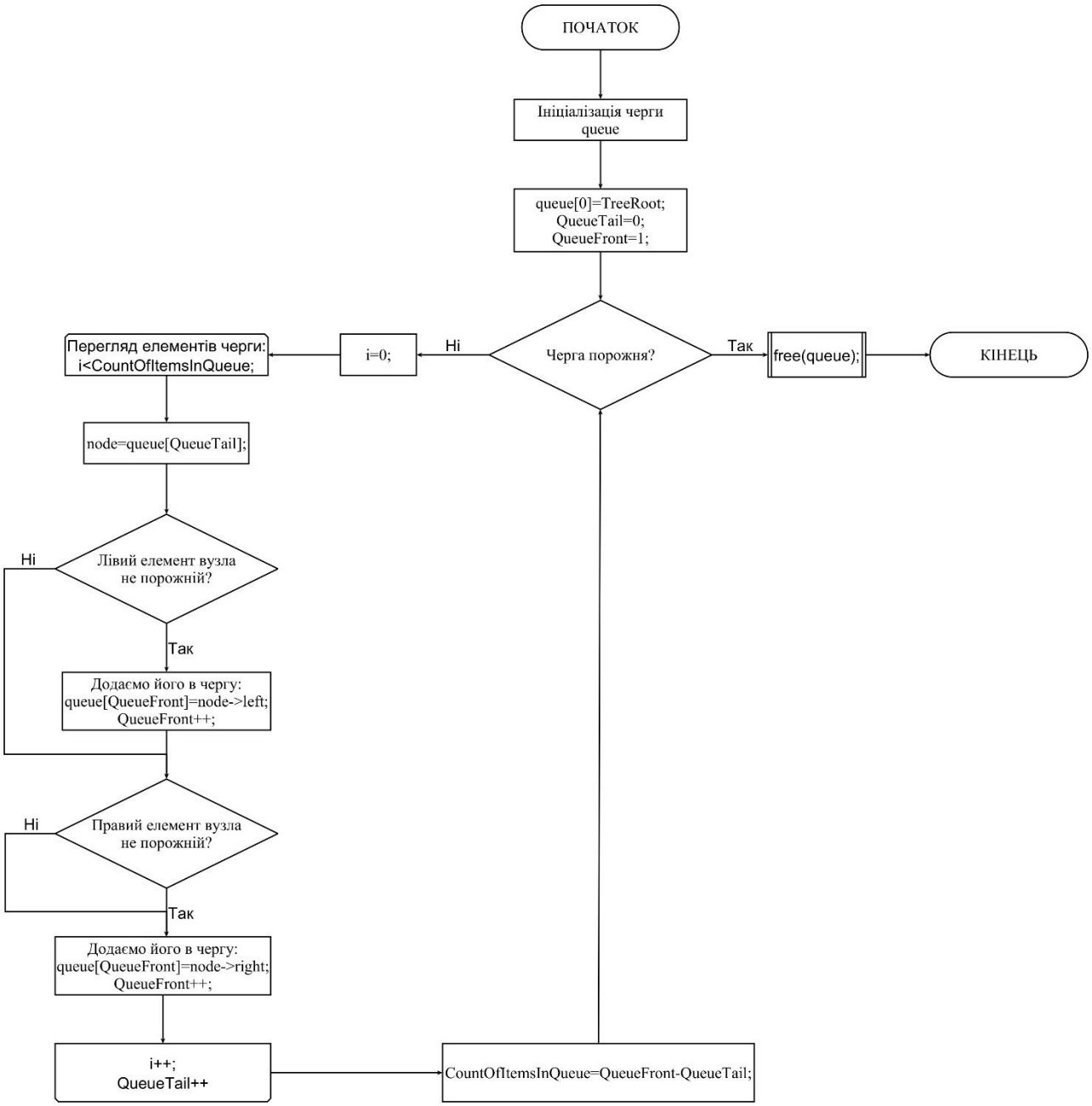
Алгоритм ротації вузлів вправо



Алгоритм ротації вузлів вліво



Нерекурсивний алгоритм обходу бінарного та червоно-чорного дерев



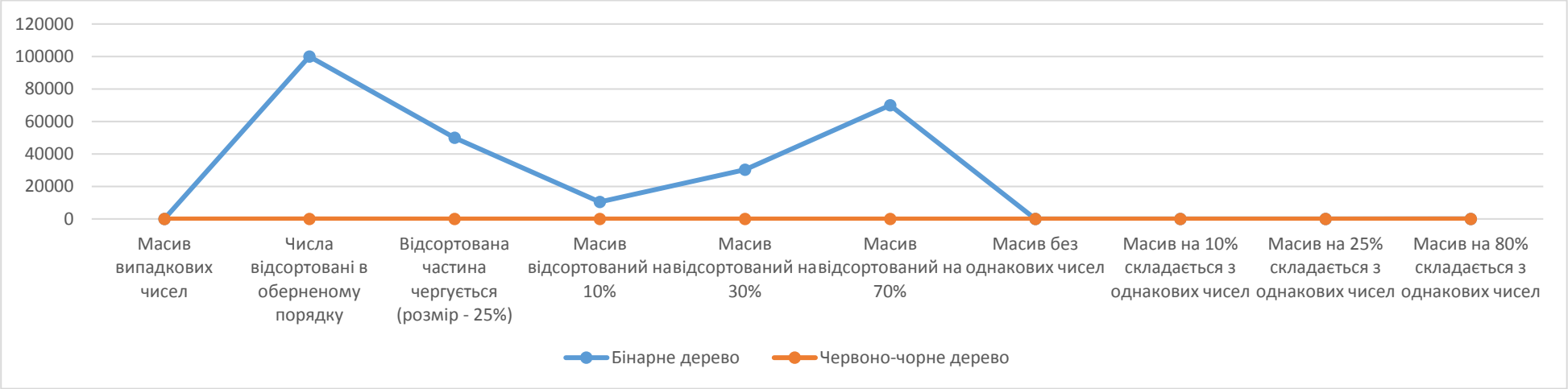
Складність розроблених алгоритмів

Назва	Випадок	Витрата пам’яті	Пошук	Вставка	Видалення
Бінарне дерево	Середній	O(n)	O(log n)	O(log n)	O(log n)
	Найгірший	O(n)	O(n)	O(n)	O(n)
Червоно-чорне дерево	Середній	O(n)	O(log n)	O(log n)	O(log n)
	Найгірший	O(n)	O(log n)	O(log n)	O(log n)

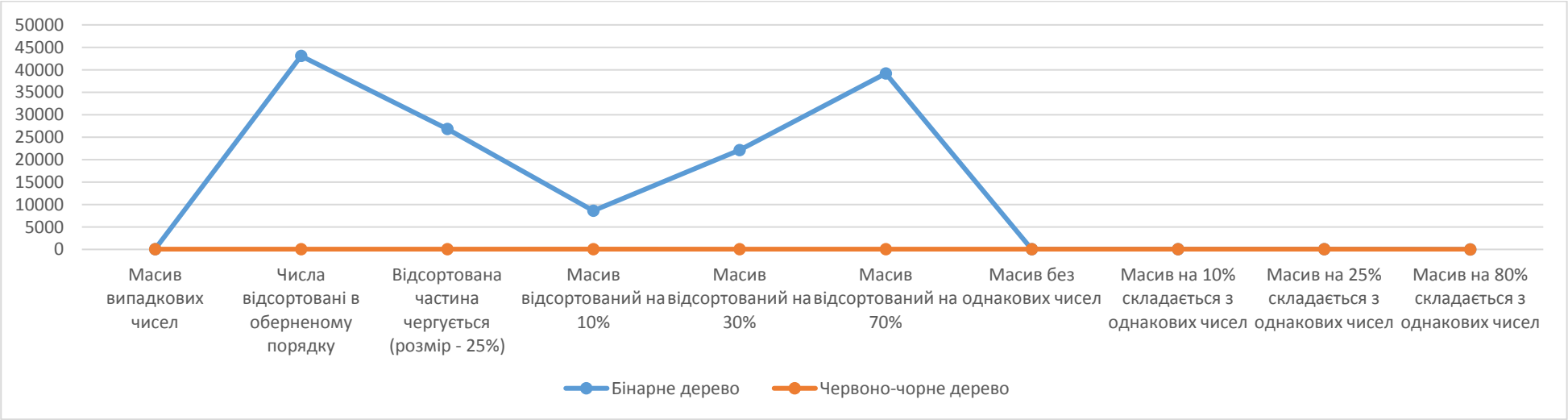
Порівняння ефективності бінарного та червоно-чорного дерев при кількості елементів 100000

(час вказаний у мілісекундах)

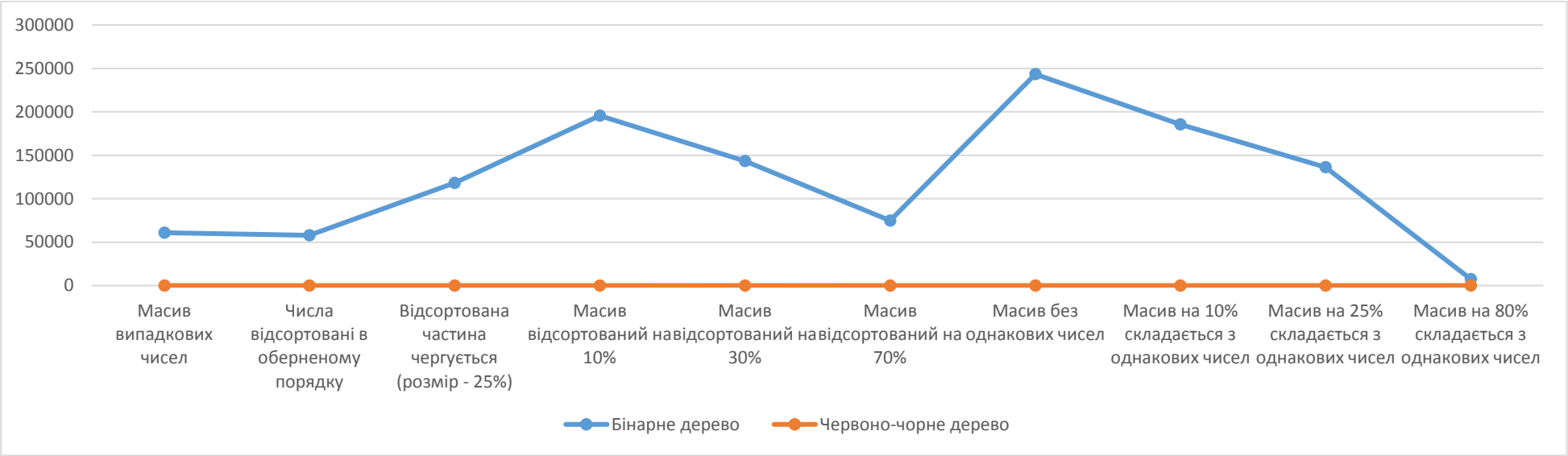
Висота дерев (максимальна довжина списків)



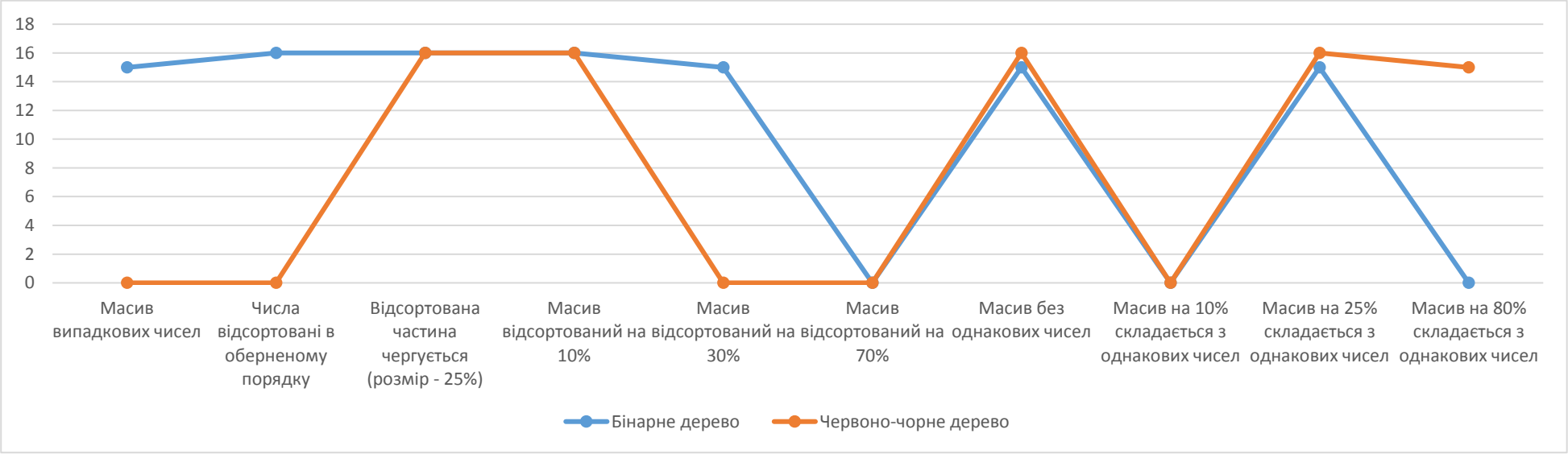
Час додавання всіх елементів



Час пошуку всіх елементів

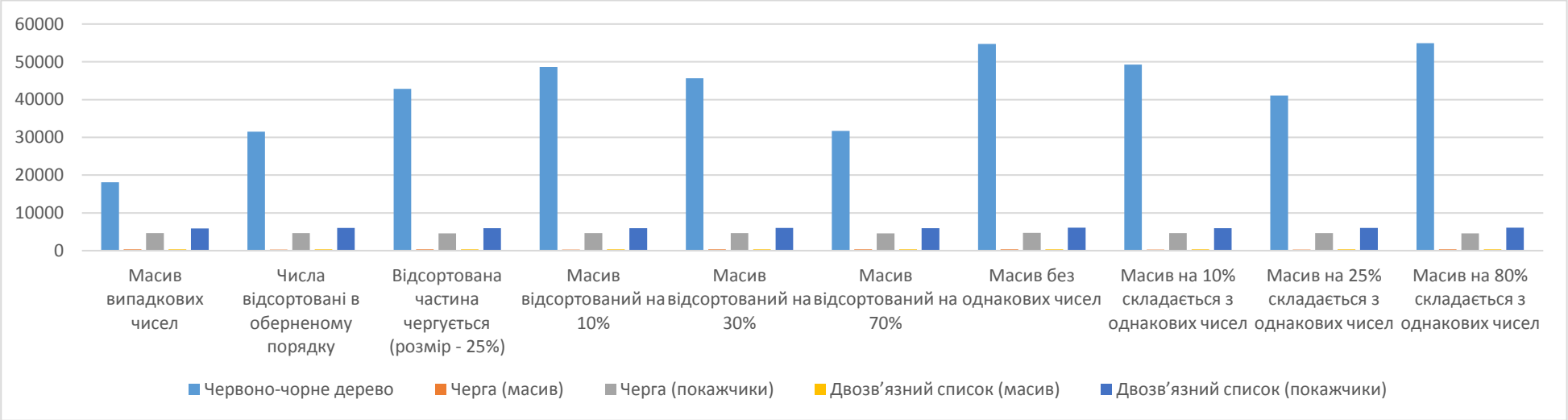


Час звільнення пам’яті

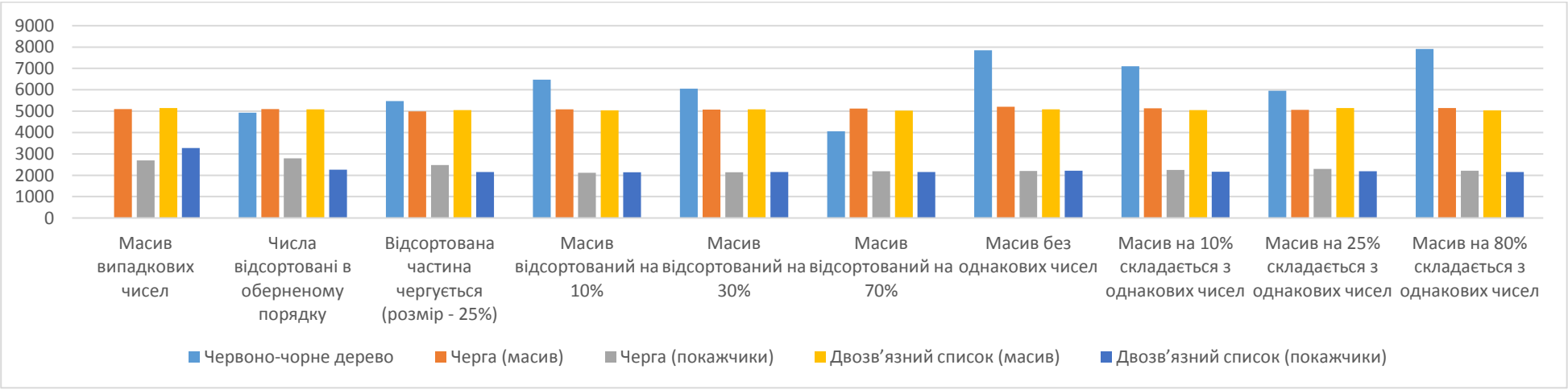


Порівняння ефективності червоно-чорного дерева, черги та двозв’язного лінійного списку з використанням даних різного характеру в кількості 50 млн чисел (час вказаний у мілісекундах)

Операція додавання



Операція видалення



4. Висновок. Бінарні та червоно-чорні дерева доцільно використовувати для збереження лише оригінальних (не однакових даних). У процесі виконання даної роботи нами було помічено, що рекурсивні алгоритми виконання основних операцій з бінарними деревами (додавання, пошук та звільнення виділеної пам’яті) спричинюють переповнення стеку викликів операційної системи та некоректне завершення програми при спробі обробки приблизно 50000 елементів. Тому ми реалізували нерекурсивні рішення, що підвищило їх продуктивність та уможливило обробку значно більшого обсягу чисел.

Порівнявши швидкодію виконання операції додавання в бінарне та червоно-чорне дерева ми помітили, що найбільш швидкодіючою структурою даних є друга. Згідно з вище наведеним графіком, швидкість виконання операції додавання в бінарне дерево залежить від характеру даних: відсортовані числа додаються швидше, а однакові опускаються. Було помічено, що червоно-чорне дерево є найменш залежним від типу вхідних даних при виконанні операцій додавання та звільнення виділеної пам’яті. Висота дерев, збалансованість та характер даних в них можуть впливати на швидкодію пошуку. Червоно-чорне дерево при будь-яких даних, навіть відсортованих, є значно нижчим за бінарне, що дозволяє знаходити інформацію за найменшу кількість кроків.

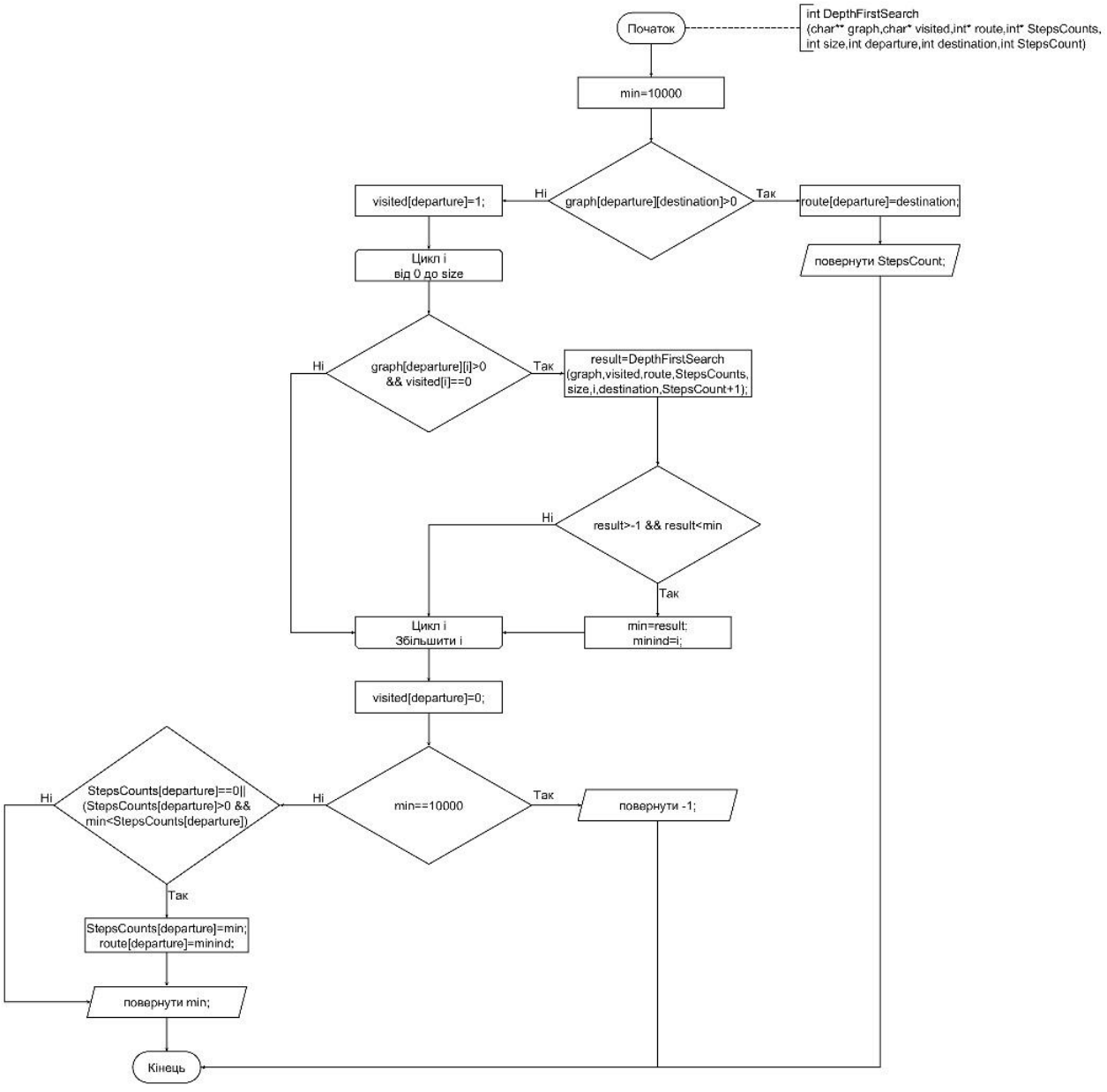
В рамках даної роботи було виконано порівняння ефективності червоно-чорного дерева та абстрактних структур даних «черга» і «двозв’язний лінійний список» на значно більшій кількості елементів, ніж попередні дослідження. Подібне випробування не виконувалось з бінарним деревом через дуже великі часові витрати на додавання елементів. Було встановлено, що червоно-чорне дерево є найменш швидкодіючим у порівнянні з названими структурами даних.

ПРАКТИЧНІ РОБОТИ №8-10. Алгоритми пошуку в графі. Пошук в ширину та в глибину. Знаходження найкоротших шляхів. Каркас мінімальної ваги.

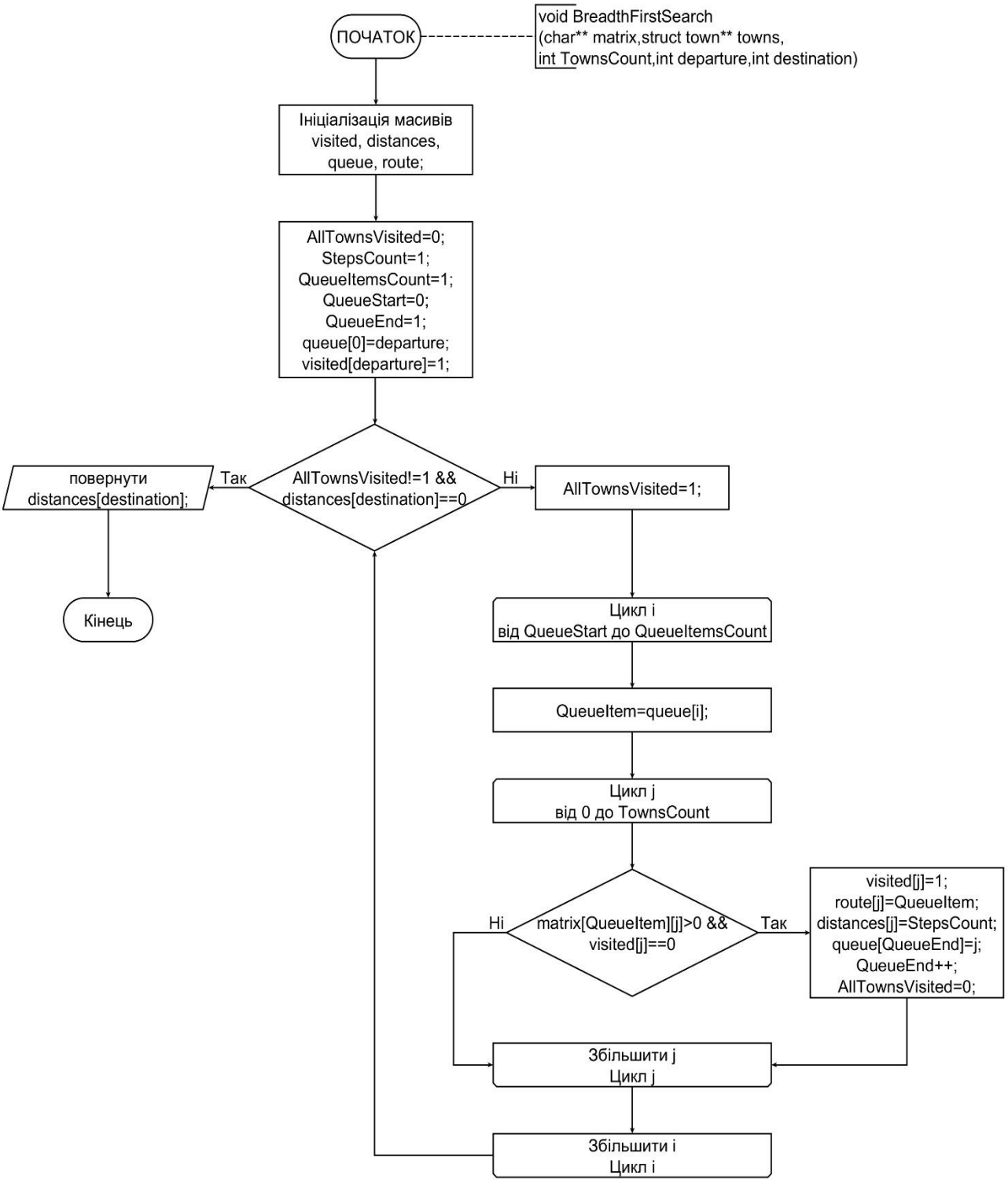
1. Постановка задачі: написати програмні реалізації алгоритмів пошуку в ширину та в глибину для знаходження мінімальної кількості кроків між двома точками; реалізувати алгоритми обчислення найменшої відстані та протестувати всі рішення на випадкових графах різного ступеня зв'язності, а також на прикладі реального графу у вигляді карти частини території України; зробити код двох алгоритмів пошуку каркасу графа мінімальної ваги.

2. Схеми реалізованих алгоритмів

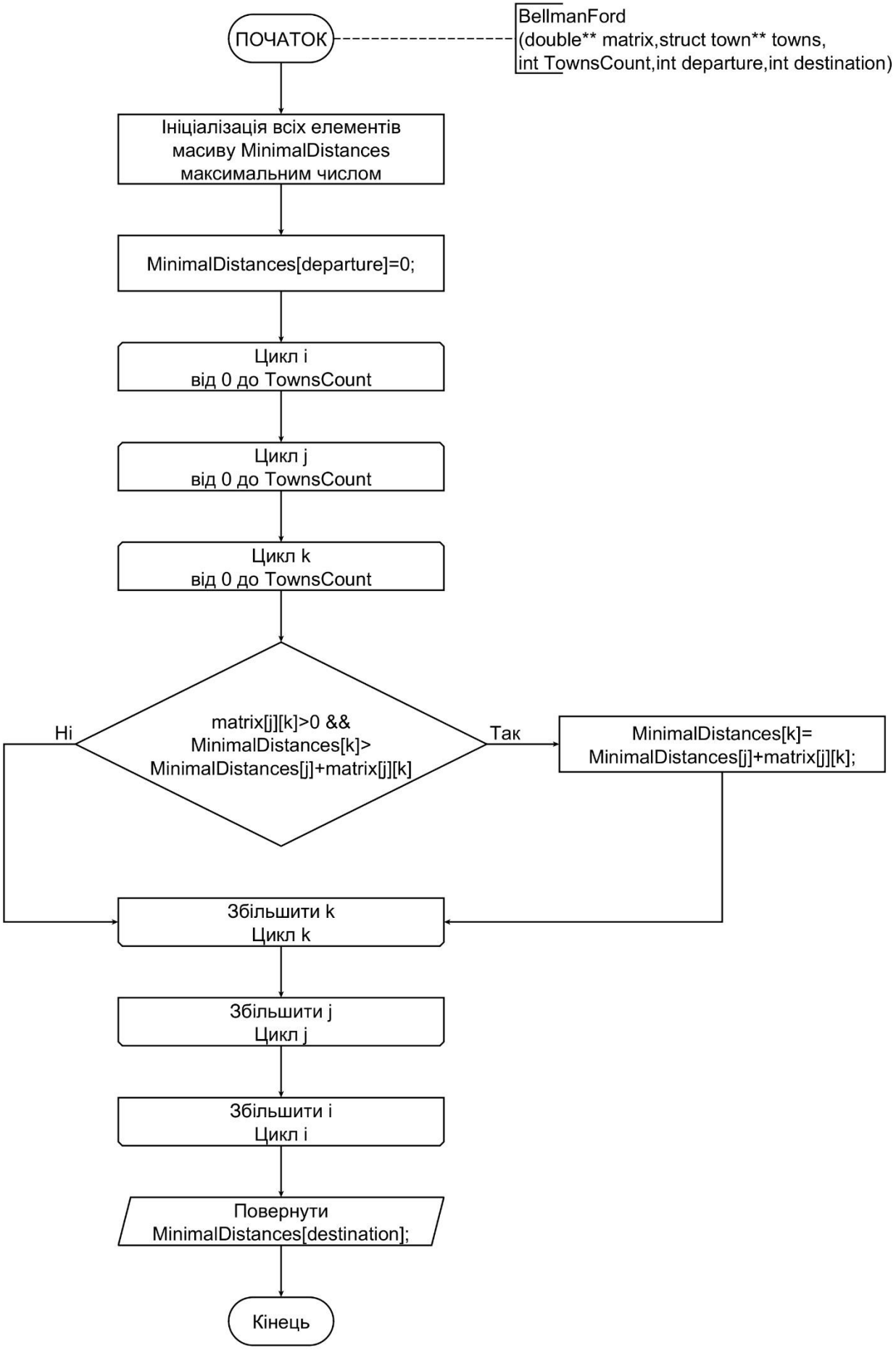
Пошук в глибину (рекурсія)



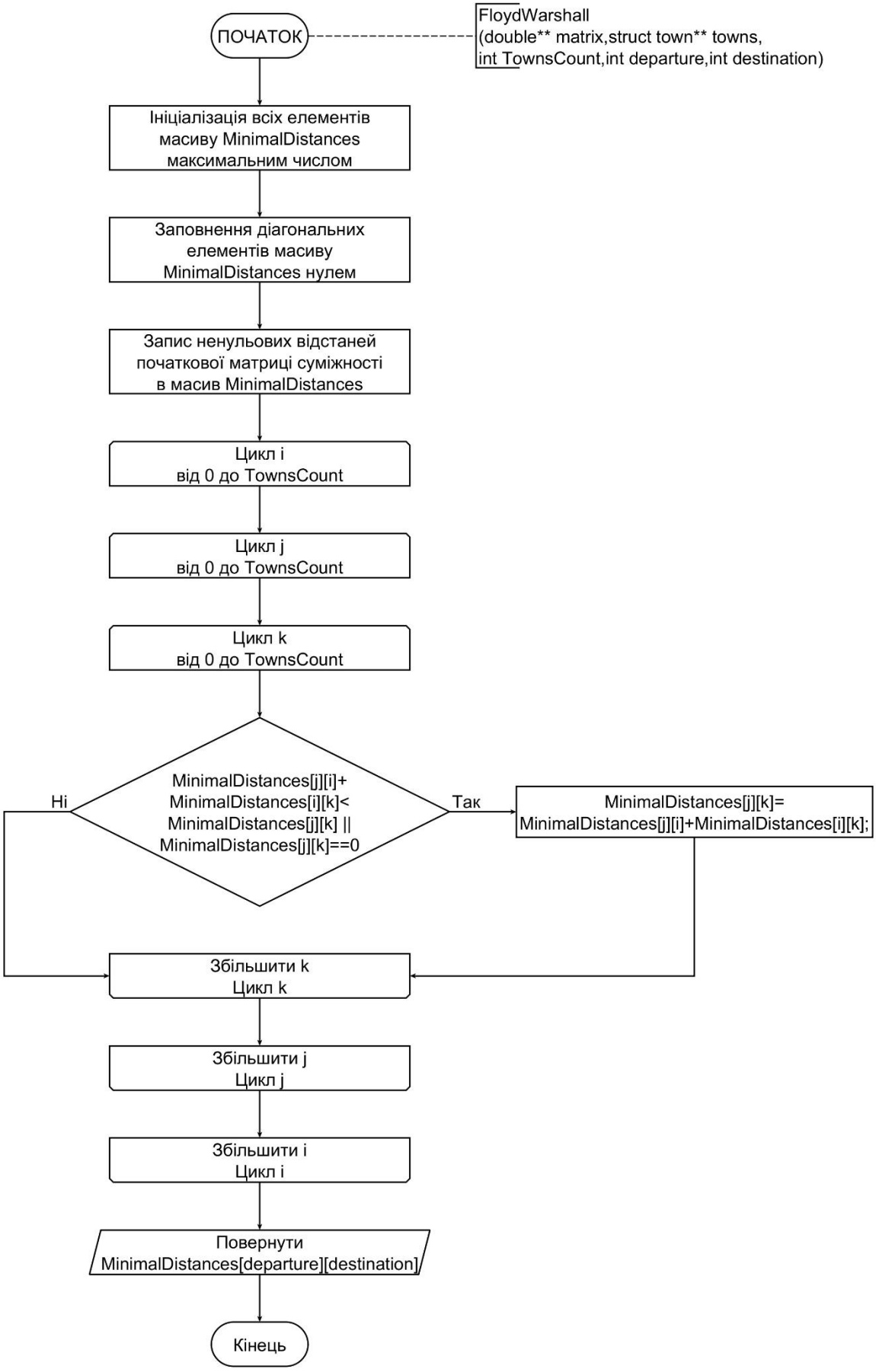
Пошук в ширину



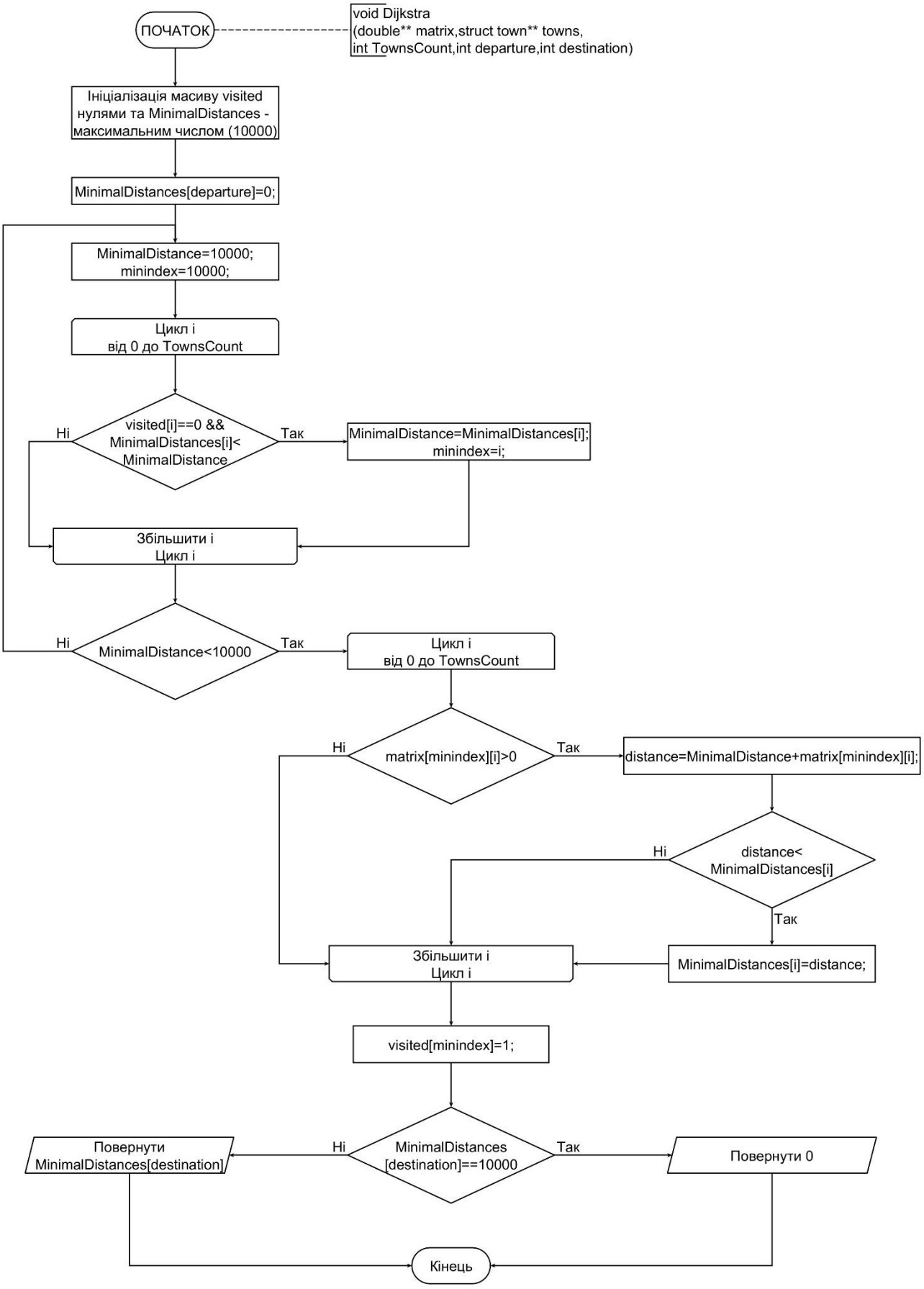
Алгоритм Беллмана-Форда



Алгоритм Флойда-Уоршелла



Алгоритм Дейкстри



3. Робота з реальним графом.

На основі карти України обрати частину території, яка включає в себе населений пункт проживання студента, населений пункт його рідних (не співпадає з попереднім) та ще декілька населених пунктів навколо них. На основі виділеної території створити базовий граф, вершини якого – населені пункти, ребра – дороги між ними. Кількість вершин повинна бути від 15 до 20, кількість ребер від 30 . Вагами ребер необхідно взяти:

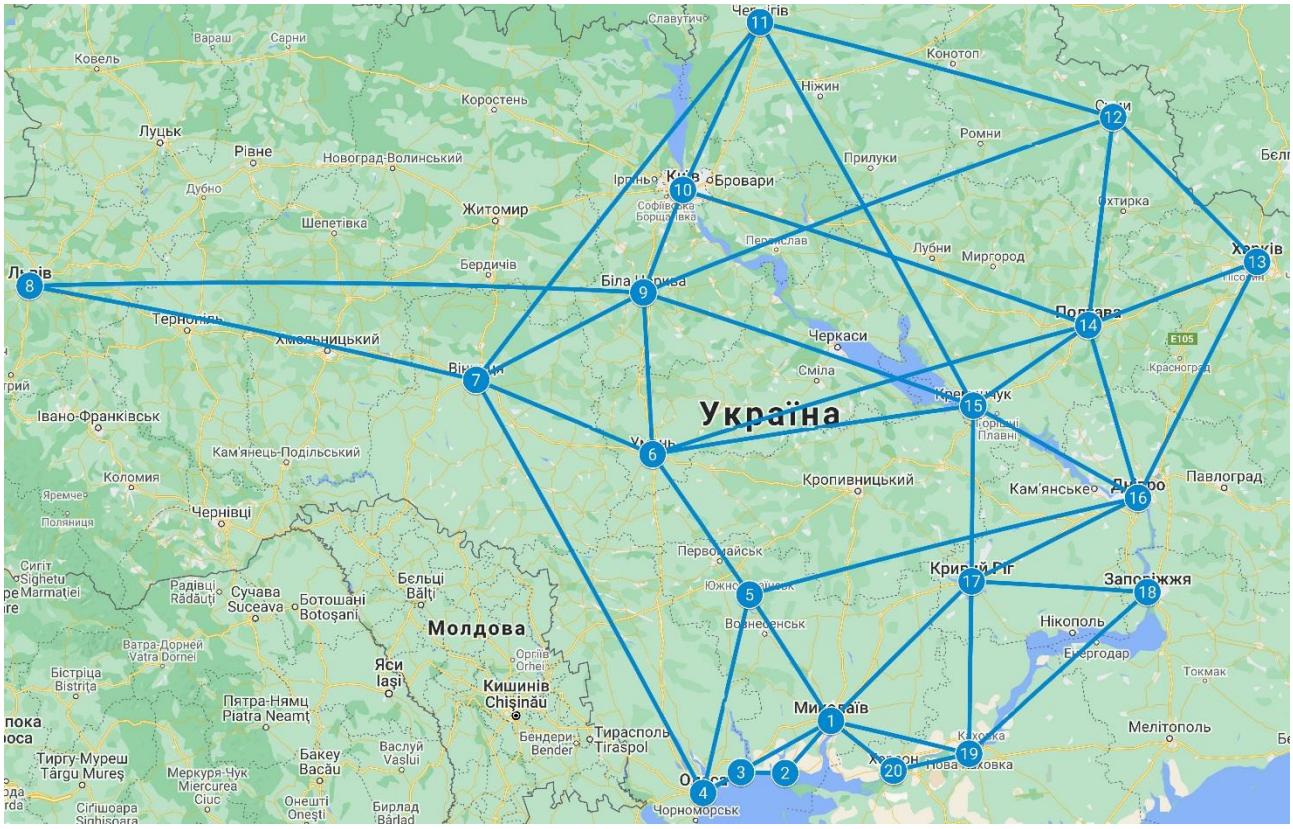
a) довжину шляху між населеними пунктами в км,

b) тип дороги, ставлячи йому відповідний пріоритет.

Для кожного населеного пункту визначити його координати та кількість населення.

Скріншот частини території України, що застосовується в роботі.

Базовий граф:



Розв'язок задач.

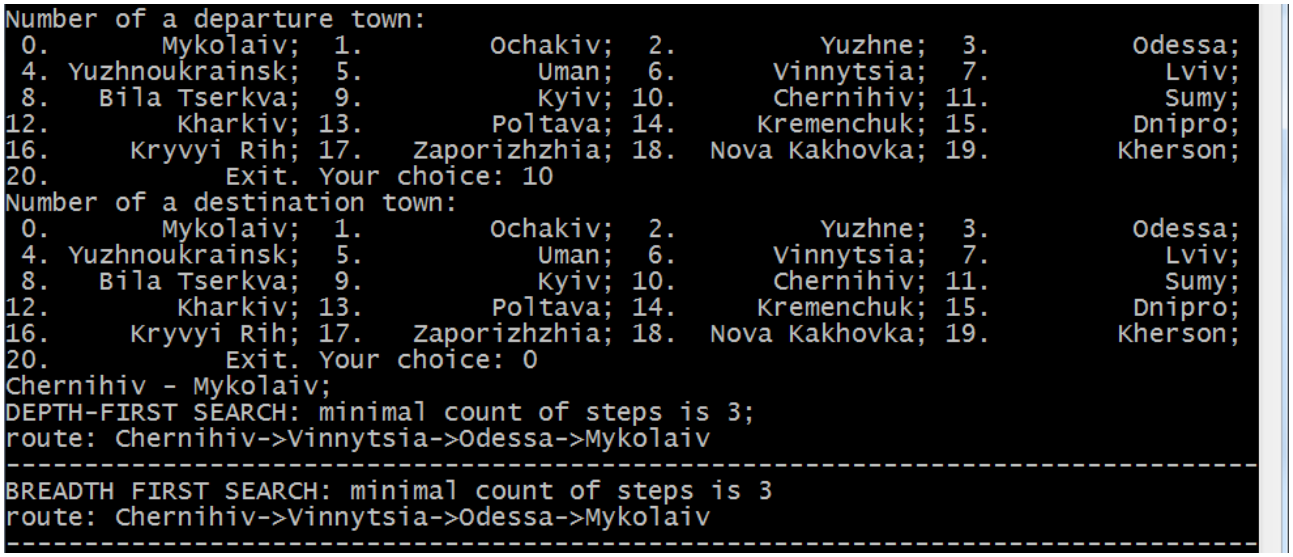
А) За допомогою одного з алгоритмів обходу графів обійти всі населені пункти на виділеній територій. Початком вважати населений пункт проживання студента, кінцевою точкою – пункт проживання його рідних. На довжину шляху та тип дороги не зважати.

Для обходу графу використовується рекурсивний алгоритм в глибину.

```
C:\Windows\system32\cmd.exe
1. Visit all towns starting the route from Mykolaiv;
2. Find a route without taking a distance into account;
3. Find the shortest route;
4. Show towns from the west to the east;
5. Show towns from the north to the south;
6. Show towns according to the population;
7. Test random graphs;
8. Exit.
Your choice: 1
Mykolaiv->Ochakiv->Yuzhne->Odessa->Yuzhnoukrainsk->Uman->Vinnytsia->Lviv->Bila T
serkva->Kyiv->Chernihiv->Sumy->Kharkiv->Poltava->Kremenchuk->Dnipro->Kryvyi Rih-
>Zaporizhzhia->Nova Kakhovka->Kherson
```

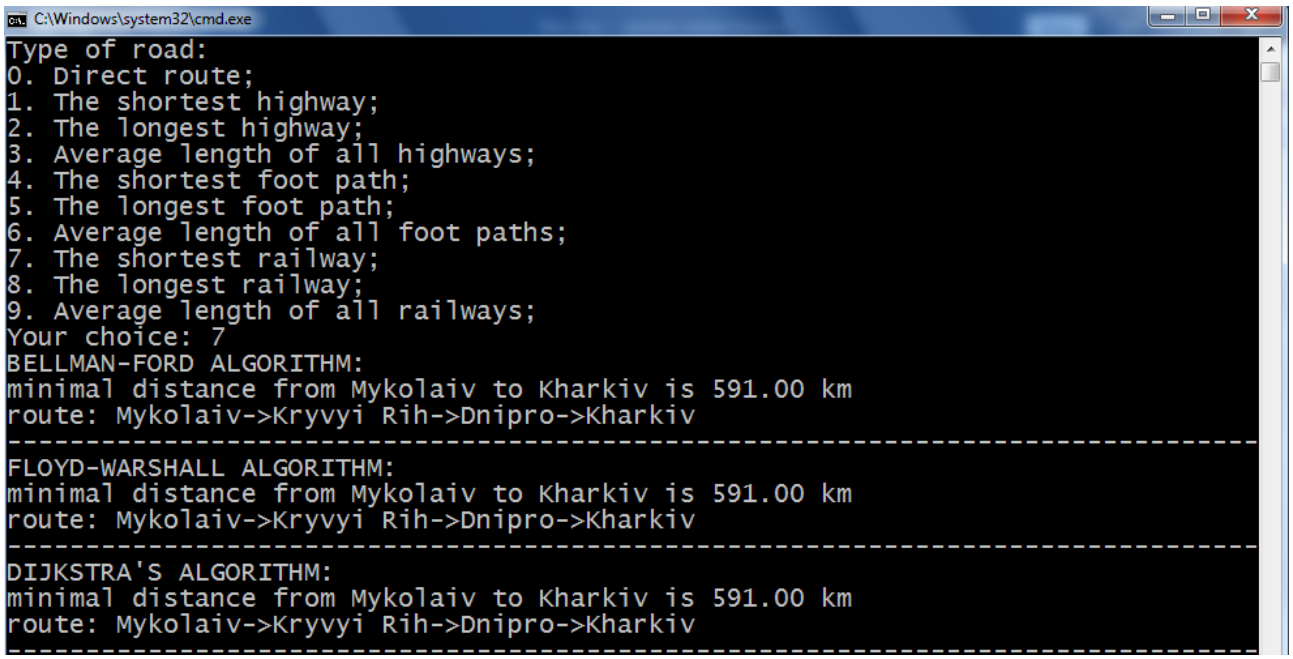
Б) За допомогою одного з алгоритмів знайти шлях від населеного пункту проживання студента до пункту проживання його рідних. На довжину шляху та тип дороги не зважати.

Для пошуку найменшої кількості кроків використовуються алгоритми пошуку в ширину та глибину:



В) За допомогою одного з алгоритмів знайти найкоротший шлях для двох випадків обирання критерію проходження (вага a) та b)) від населеного пункту проживання студента до пункту проживання його рідних. Порівняти результати.

Для вирішення цього завдання нами використовуються три алгоритми: Дейкстри, Форда-Беллмана та Флойда-Уоршелла:



З Одеси немає залізничного шляху до Очакову:

```
Type of road:
0. Direct route;
1. The shortest highway;
2. The longest highway;
3. Average length of all highways;
4. The shortest foot path;
5. The longest foot path;
6. Average length of all foot paths;
7. The shortest railway;
8. The longest railway;
9. Average length of all railways;
Your choice: 9
BELLMAN-FORD ALGORITHM:
there is no route from Odessa to Ochakiv.
FLOYD-WARSHALL ALGORITHM:
there is no route from Odessa to Ochakiv.
DIJKSTRA'S ALGORITHM:
there is no route from Odessa to Ochakiv.
```

Г) Зробити сортування населених пунктів на основі відомих алгоритмів за їх координатами із заходу на схід, та кількістю населення з більшого к меншому.

Задача вирішувалась за допомогою сортування вставками. Населені пункти із заходу на схід:

```
Your choice: 4
 0.          Lviv;  1.      Vinnytsia;  2.   Bila Tserkva;  3.          Uman;
 4.          Kyiv;  5.        Odessa;  6.        Yuzhne;  7. Yuzhnoukrainsk;
 8.      Chernihiv;  9.       Ochakiv; 10.      Mykolaiv; 11.       Kherson;
12.  Nova Kakhovka; 13.     Kryvyi Rih; 14.    Kremenchuk; 15.       Poltava;
16.          Sumy; 17.        Dnipro; 18.   Zaporizhzhia; 19.       Kharkiv;
```

Населені пункти з півдня на північ:

```
Your choice: 5
 0.         Odessa;  1.       Ochakiv;  2.        Yuzhne;  3.       Kherson;
 4.  Nova Kakhovka;  5.      Mykolaiv;  6. Yuzhnoukrainsk;  7.   Zaporizhzhia;
 8.     Kryvyi Rih;  9.       Dnipro; 10.          Uman; 11.     Kremenchuk;
12.      Vinnytsia; 13.       Poltava; 14.   Bila Tserkva; 15.          Lviv;
16.        Kharkiv; 17.          Kyiv; 18.          Sumy; 19.      Chernihiv;
```

Населені пункти розташовані за кількістю населення:

```
Your choice: 6
 0.          Kyiv;  1.       Kharkiv;  2.        Odessa;  3.        Dnipro;
 4.   Zaporizhzhia;  5.          Lviv;  6.     Kryvyi Rih;  7.      Mykolaiv;
 8.      Vinnytsia;  9.       Kherson; 10.       Poltava; 11.     Chernihiv;
12.          Sumy; 13.    Kremenchuk; 14.   Bila Tserkva; 15.          Uman;
16.  Nova Kakhovka; 17. Yuzhnoukrainsk; 18.        Yuzhne; 19.       Ochakiv;
```

Д) Для кожного попереднього пункту описати обраний алгоритм, його особливості, обґрунтувати вибір цього алгоритму та визначити його складність.

Алгоритм	**Складність**
Пошук в глибину	O(m)
Пошук в ширину	O(V+E)
Алгоритм Дейкстри	O(n log n + m log n)=O(m log n)
Алгоритм Форда-Беллмана	Найгірший випадок: O(VE); найкращий: O(E)
Алгоритм Флойда-Уоршелла	$O(V^3)$

Для пошуку найменшої кількості кроків з однієї вершини в іншу без урахування відстані нами було реалізовано алгоритми в глибину та ширину. Пошук в глибину є рекурсивним і основна його суть полягає в тому, що ми рухаємось в глиб графу доки це можливо і фактично здійснюємо вибір серед всіх варіантів шляхів, вибираючи найкоротший. Він є найменш ефективним і дуже повільно працює на сильно зв'язних графах, але добре підходить для дерев (незв'язних графів). Пошук в ширину є більш швидким в усіх випадках, оскільки заснований на ідеї динамічного програмування. Його суть полягає в тому, що ми дивимось навколо, запам'ятовуючи найменшу кількість кроків з кожної вершини до сусідніх.

Для пошуку найкоротших шляхів з урахуванням відстані ми використали три алгоритми: Дейкстри, Форда-Беллмана та Флойда-Воршелла.

Алгоритм Флойда-Воршелла порівнює всі можливі шляхи в графі між кожною парою вершин. Він виконується за $\Theta(|V|^3)$ порівнянь. Це доволі примітивно, враховуючи, що в графі може бути до $\Omega(|V|^2)$ ребер, і кожну комбінацію буде перевірено. Він виконує це шляхом поступового поліпшення оцінки по найкоротшому шляху між двома вершинами, поки оцінка не стає оптимальною.

Розгляньмо граф G з ребрами V, пронумерованими від 1 до N. Крім того розгляньмо функцію shortestPath(i, j, k), яка повертає найкоротший шлях від і до j, використовуючи вершини з множини {1,2,…,k} як внутрішні у шляху.

Тепер, маючи таку функцію, нам потрібно знайти найкоротший шлях від кожного i до кожного j, використовуючи тільки вершини від 1 до k + 1.

Для кожної з цих пар вершин, найкоротший шлях може бути або (1)-шлях, у якому є тільки вершини з множини {1, …, k}, або (2)- шлях, який проходить від i до k + 1 а потім від k + 1 до j. Найкоротший шлях від i to j that only uses vertices 1 через k визначається функцією shortestPath(i, j, k), і якщо є коротший шлях віді до (k + 1 до j), то довжина цього шляху буде сумою(конкатенацією) найкоротшого шляху віді до k + 1 (використовуючи вершини{1, …, k}) і найкоротший шлях від k + 1 до j (також використовуючи вершини з {1, …, k}).

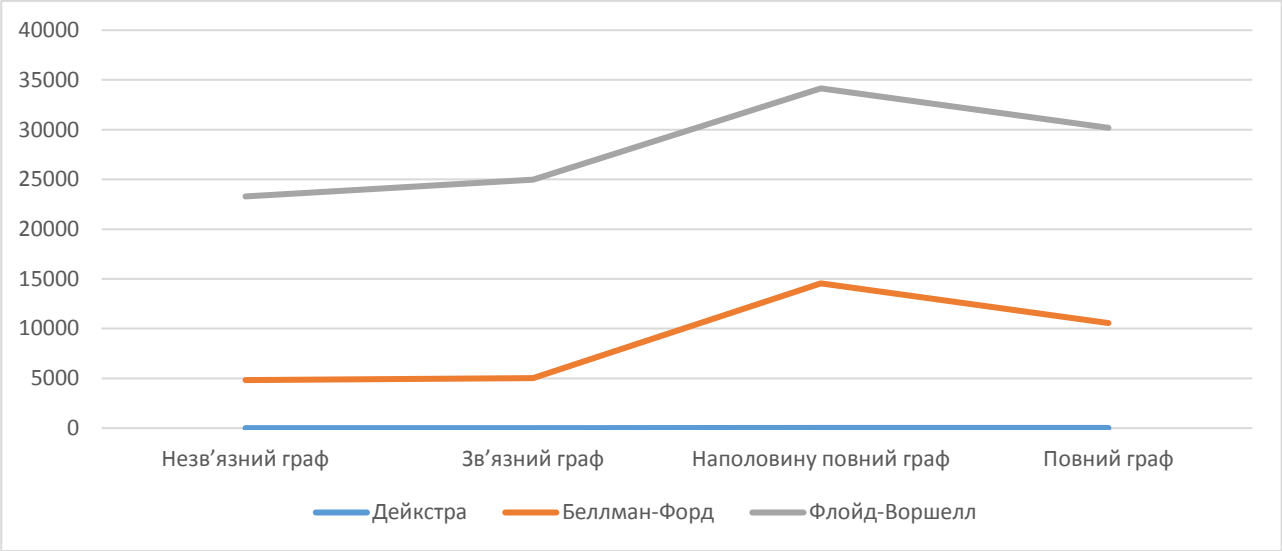
Алгоритм Форда-Беллмана представляє з себе кілька фаз. На кожній фазі проглядаються всі ребра графа, і алгоритм намагається справити релаксацію (relax, ослаблення) уздовж кожного ребра (u, v) ваги w(u, v). Релаксація вздовж ребра — це спроба поліпшити значення v.d значенням v.u + w(u, v). Фактично це означає, що ми намагаємося поліпшити значення для вершини v, користуючись ребром (u, v) і поточним значенням для вершини u. Стверджується, що достатньо G. V-1 фази алгоритму, щоб коректно порахувати довжини всіх найкоротших шляхів у графі (цикли негативної ваги відсутні). Для недосяжних вершин відстань v. d залишиться нескінченністю.

Найпростіша реалізація алгоритму Дейкстри потребує $O(V^2)$ дій. У ній використовується масив відстаней та масив позначок. На початку алгоритму відстані заповнюються великим додатнім числом (більшим максимального можливого шляху в графі), а масив позначок заповнюється нулями. Потім відстань для початкової вершини вважається рівною нулю і запускається основний цикл.

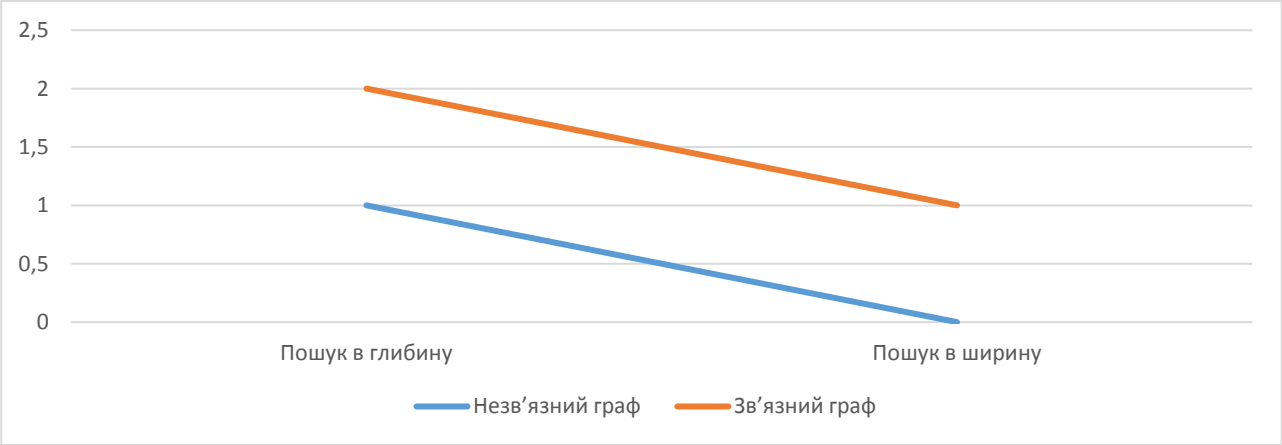
На кожному кроці циклу ми шукаємо вершину з мінімальною відстанню і прапором рівним нулю. Потім ми встановлюємо в ній позначку 1 і перевіряємо всі сусідні з нею вершини. Якщо в ній відстань більша, ніж сума відстані до поточної вершини і довжини ребра, то зменшуємо його. Цикл завершується коли позначки всіх вершин стають рівними 1.

Е) Тест швидкодії роботи всіх алгоритмів на кількох випадкових матрицях суміжності розміром 1000 на 1000 (час – у мілісекундах). Вказано середні результати серед 10 проведених випробувань пошуку відстані.

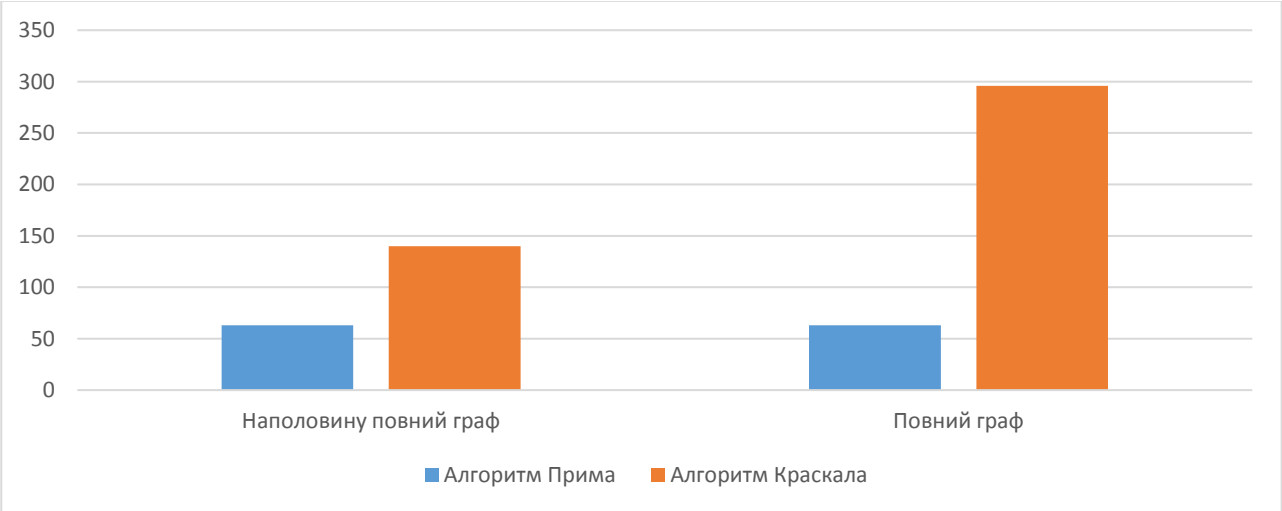
Пошук найкоротшої відстані



Пошук найменшої кількості кроків



Пошук остовного дерева (каркасу графа) мінімальної ваги



Загальний висновок. У процесі виконання роботи ми реалізували всі основні популярні алгоритми обходу та пошуку найкоротшої відстані і перевірили коректність їх роботи як на реальному невеликому графі (20 вершин та 40 ребер), який є зображенням частини території України, так і на випадкових графах різного ступеня зв’язності. Оскільки алгоритм пошуку в глибину дуже повільно працює на сильно зв’язних графах, експерименти з такими графами не проводились. Реалізовані нами алгоритми пошуку каркасу графа некоректно працюють з незв’язними графами, тому надано результати швидкодії роботи лише для наполовину повного та повного графів. У процесі дослідження було встановлено, що найбільш швидкодіючим алгоритмом пошуку найменшої кількості кроків з однієї вершини в іншу є пошук в ширину, а найбільш ефективним способом знаходження найменшої відстані є алгоритм Дейкстри. Найгіршим з досліджених алгоритмів є метод Флойда-Воршелла, який найкраще всього підходить для орієнтованих графів. Алгоритм Форда-Беллмана може працювати з від’ємними вагами, на відміну від найшвидшого алгоритму Дейкстри.