

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Кафедра автоматизації проектування
енергетичних процесів і систем

КУРСОВА РОБОТА

з дисципліни «Алгоритмізація та програмування 2. Процедурне програмування»
(назва дисципліни)

на тему:
«Використання алгоритмів в практичних задачах»

Студента групи TP-12
спеціальність 122 «Комп'ютерні науки»
спеціалізація «Комп'ютерний моніторинг та геометричне моделювання процесів і систем»

Каркушевський В.Л.
(прізвище та ініціали)

Керівник ас. Москаленко Ю.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Національна оцінка _____

Кількість балів: _____ Оцінка: ECTS _____

Члени комісії

(підпис)

ас. Москаленко Ю.В.
(вчене звання, науковий ступінь, прізвище та ініціали)

(підпис)

доц., к.т.н. Шаповалова С.І.
(вчене звання, науковий ступінь, прізвище та ініціали)

(підпис)

доц., к. військ. н. Онисько А. І.
(вчене звання, науковий ступінь, прізвище та ініціали)

Київ - 2022 рік

**Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”**

Факультет ТЕПЛОЕНЕРГЕТИЧНИЙ ФАКУЛЬТЕТ

(повна назва)

Кафедра автоматизації проектування енергетичних процесів та систем

(повна назва)

Освітньо-кваліфікаційний рівень бакалавр

спеціальність 122 «Комп'ютерні науки»

(шифр і назва)

спеціалізація «Комп'ютерний моніторинг та геометричне моделювання процесів і систем»

(шифр і назва)

**З А В Д А Н Н Я
НА КУРСОВУ РОБОТУ СТУДЕНТУ**

Каркушевський Владислав Леонідович

(прізвище, ім'я, по батькові)

1. Тема роботи «Використання алгоритмів в практичних задачах»

керівник курсової роботи –

Москаленко Юрій Володимирович, асистент

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

2. Строк подання студентом роботи –

3. Вихідні дані до проекту (роботи): мова JAVA, використання алгоритмів в практичних задачах

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання курсової роботи	Строк виконання етапів роботи	Примітка
1	Отримання завдання	1-3 тижні	
2	Вибір та дослідження методів, вибір відповідних структур даних, розробка алгоритмів	4-6-й тижні	
3	Програмна реалізація	6-10-й тижні	
4	Демонстрація першого варіанту	11-ий тиждень	
5	Тестування програми	11-ий тиждень	
6	Оформлення звіту	12-й тиждень	
7	Захист курсової	13-й тиждень	

Студент

_____ (підпис)

Керівник курсової роботи

_____ (підпис)

Каркушевський В.Л.

(прізвище та ініціали)

Москаленко Ю.В.

(прізвище та ініціали)

АНОТАЦІЯ

У результаті виконання курсової роботи розроблено програму з використанням алгоритмів в практичних задачах. Розроблено основні структури даних такі як: двозв'язний список, двійкове дерево пошуку та орієнтований граф. Розроблено завантаження даних з файлу csv. Досліджено алгоритми пошуку у ширину та глибину, алгоритм Дейкстри та розривання деяких вершин графу та перерахунок шляхів. Розроблено програмний продукт, в основі якого лежить граф, з містами України, у якому можна визначити найкоротший шлях від одного міста до іншого, визначити кількість палива яку потрібно витратити на маршрут та кількість вантажу яку потрібно розвантажити у місті.

Код створений на мові програмування Java . Програма розроблена у інтегрованому середовищі розробки програмного забезпечення JetBrains IntelliJ IDEA 2021.

ANNOTATION

As a result of the course work, a program was developed using algorithms in practical problems. Basic data structures such as a two-linked list, a binary search tree, and an oriented graph have been developed. Data download from csv file developed. The algorithms of search in width and depth, Dijkstra's algorithm and breaking of some vertices of the graph and recalculation of ways are investigated. A software product based on the graph has been developed with the cities of Ukraine, in which you can determine the shortest way from one city to another, determine the amount of fuel to be spent on the route and the amount of cargo to be unloaded in the city.

The code is created in the Java programming language. The program is developed in the integrated software development environment JetBrains IntelliJ IDEA 2021.

ЗМІСТ

Вступ.....	7
1. Теоретичні відомості.....	8
1.1. Алгоритми та їх особливості.....	8
1.1.1 Властивості та способи описання алгоритму.....	8
1.1.2 Блок схема та її побудова.....	9
1.1.3 Часова складність.....	10
1.2. Структури даних.....	11
1.2.1 Ієрархія структури даних.....	11
1.2.2 Примітивні структури даних.....	12
1.2.3 Масиви.....	13
1.2.4 Стек.....	14
1.2.5 Черга.....	15
1.2.6 Зв'язні списки.....	16
1.2.7 Дерево.....	17
1.3. Граф та алгоритми у графові.....	20
1.3.1 Граф.....	20
1.3.2 Пошук в ширину(BFS) та в глибину(DFS).....	22
1.3.3 Алгоритм Дейкстри.....	24
2. Реалізація програмного продукту.....	25
2.1. DoublyLinkedList.....	25
2.2. Queue.....	26
2.3. BST.....	27
2.4. Vertex.....	28
2.5. Edge.....	29
2.6. Graph.....	29
2.7. BreadthFirstSearch.....	31
2.8. DepthFirstSearch.....	31
2.9. Dijkstra.....	32
2.10. ReadInfoForFile.....	33

2.11. CarTask	33
2.12. Main	34
3. Приклад роботи з продуктом.....	35
Висновки.....	37
Список використаних джерел.....	38
Додатки	40
Додаток 1. Код програми.....	40
Додаток 2. Файл info.csv.....	65

ВСТУП

У теперішній військовий час, коли Україна веде повномасштабну війну проти окупантів, дуже велику роль грають волонтери та прості люди, які допомагають ЗСУ. Більшість з них, везуть гуманітарну допомогу з-за кордону, або з західної частини України. У сьогоднішній день, є великі проблеми з паливом для автомобілів. Тому обрана тема є актуальною для всіх людей, які допомагають армії, бо допомагає знайти найкоротший шлях від міста до міста, та взнати кількість палива, яке витрачено, та кількість вантажу який вивантажений .

Завдання курсової роботи полягає у створенні програми структур та алгоритмів, та використання їх у практичних навичках. Створити граф міст, та розрахувати кількість палива, яка потрібно щоб доїхати від одного міста, до іншого. Також слід зауважити, що легша машина, витрачає менше палива, ніж завантажена машина.

Метою даної курсової роботи є реалізація програми з використанням мови програмування Java для опису алгоритмів пошуку у графі та бінарному дереві.

Ідея курсової роботи полягає у розробці графа, який відображає західну частину України, де вершини графа – це міста, а ребра – це дороги між містами. Розроблено алгоритм Дейкстри та власне завдання, яке дозволяє спочатку знайти найкоротший шлях від одного міста до іншого, а тоді визначити, кількість палива витраченого на кожному проміжку, та кількість вантажу вивантаженому в кожному місті.

1. ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1. Алгоритми та їх особливості.

Алгоритм - це набір команд, які треба виконати над вхідними даними для отримання результату.

Виконавець алгоритму – це виконавець, який може виконувати всі вказівки заданого алгоритму.

1.1.1 Властивості та способи описання алгоритму

Алгоритм не тільки задає послідовність виконання операцій при вирішенні конкретного завдання, а й повинен задовольняти такі властивості [9]:

Масовість. Алгоритм повинен діяти на будь яких елементах з множини допустимих значень.

Визначеність. Операції, які застосовуються в алгоритмі, не повинні мати незрозумілого значення. Порядок виконання повинен бути чітким

Результативність. Виконання алгоритму повинно приводити до правильного результату.

Формальність. Будь – який виконавець, який здатний виконувати вказівки алгоритму може виконати завдання і отримати результат алгоритму.

Алгоритм вважається коректним, якщо для будь-якого допустимого значення він видає правильний результат і закінчує роботу. Якщо алгоритм видає помилкове значення або не завершує свою роботу, то він некоректний.

Існують такі способи описання алгоритмів:

- Словесний
- Графічний
- Блок схема
- Псевдокод
- Мова програмування

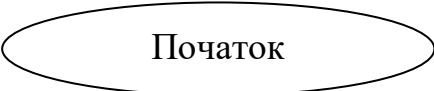
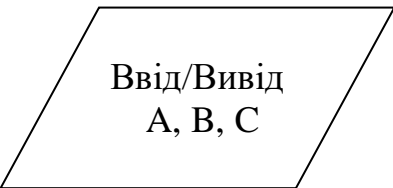
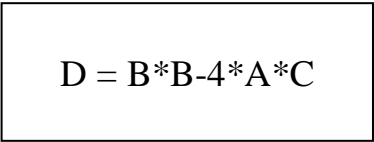
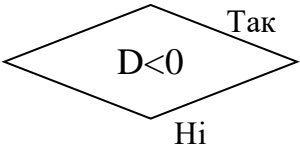
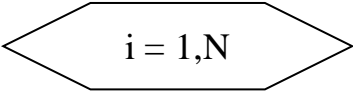

1.1.2 Блок схема та її побудова

Блок-схема алгоритму – це графічне представлення структури алгоритму, де кожний крок зображується у вигляді блоків.

Існують правила зображення блок-схем алгоритмів:

- Кожен алгоритм має початок та кінець.
- Кожна команда алгоритму представляється у вигляді блоків (таблиця 1.1).
- Блоки з'єднуються між собою лініями або стрілками, які вказують порядок виконання дій.

Таблиця 1.1. Основі блоки побудови блок схеми

Вигляд блоку	Призначення
	Початок алгоритму
	Блок вводу вхідних даних/Блок виведення результату
	Блок обробки інформації. В блоці пишуться формули.
	Блок умови. В цьому блоці прописується умова, і вибирається шлях .
	Заголовок циклу FOR
	Кінець алгоритму

1.1.3 Часова складність

Складність алгоритму - це спосіб його оцінки без прив'язки до реалізації мови програмування чи апаратного забезпечення [10].

Основними мірами обчислювальної складності алгоритмів є:

- Логічна складність – кількість людино-місяців, витрачених на створення алгоритму.
- Статична складність – довжина опису алгоритмів (кількість операторів).
- Часова складність – час виконання алгоритму.
- Ємнісна складність – характеризує об'єм пам'яті, необхідний для виконання алгоритму.

Складність алгоритму дозволяє визначитися з вибором ефективного алгоритму серед тих, що побудовані для розв'язання конкретної проблеми.

Для позначення оцінки складності використовують O-нотацію (рисунок 1.1) — вираз $O(f(n))$, який означає, що час виконання алгоритму зростає з тією ж швидкістю, що і функція $f(n)$ [5].

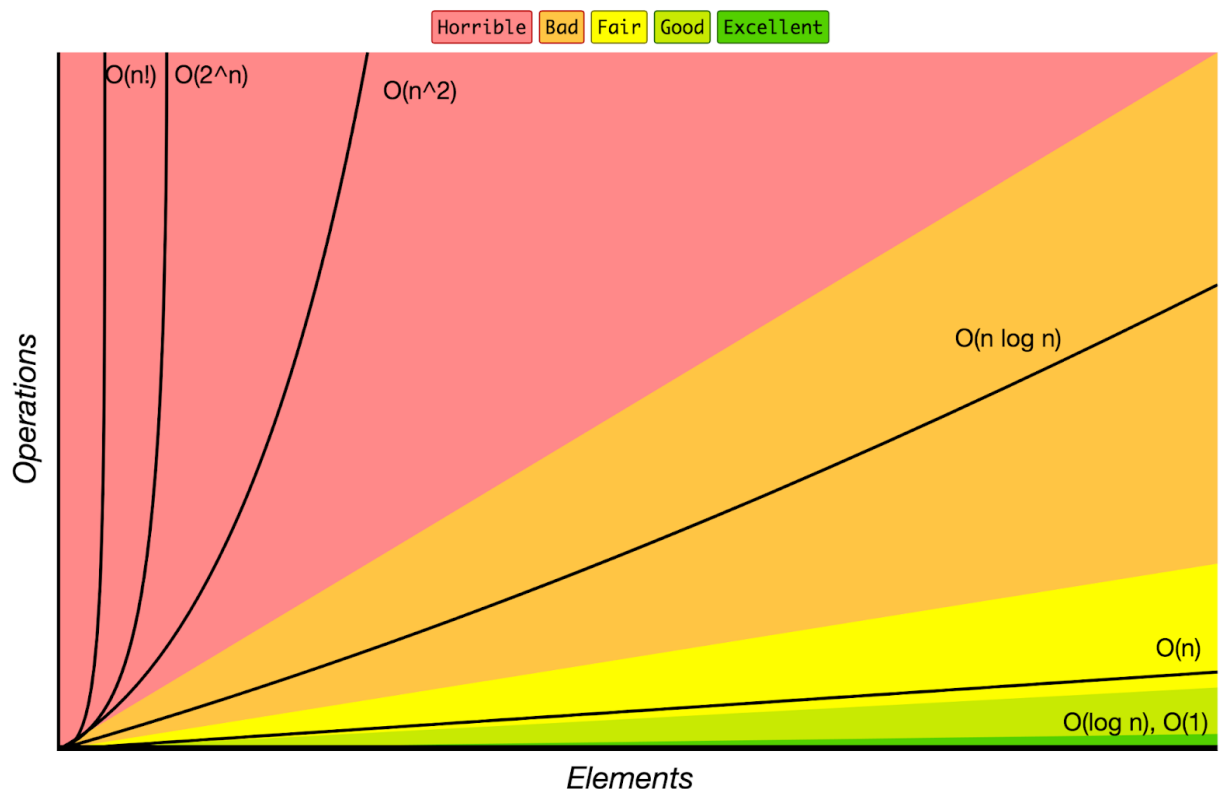


Рисунок 1.1 – Графік зростання часу алгоритму

1.2. Структури даних.

1.2.1 Ієрархія структури даних

Структура даних — це сукупність значень даних і зв'язків між ними. Структури даних дозволяють ефективно зберігати та обробляти дані. Існує ціла ієрархія структур даних (рисунок 1.2) [11], та багато різних структур даних, кожна з яких має свої переваги та недоліки. Деякі з найпоширеніших структур даних — це масиви, списки, дерева та графи [11].

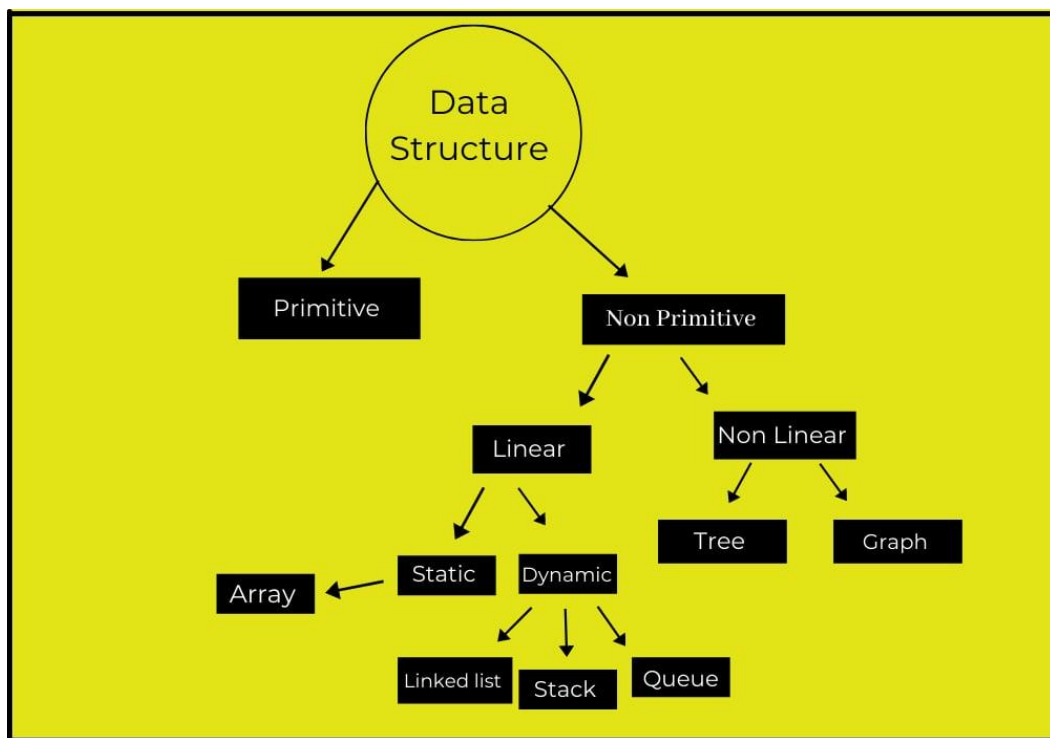


Рисунок 1.2 – Ієрархія структури даних

Існує 2 типи структури даних:

- Примітивні
- Не примітивні

Примітивні структури даних служать основою для побудови більш складних структур.

Не примітивні структури даних — це складні структури даних, в яких використовуються примітивні типи даних.

Не примітивні структури даних даних поділяються на:

- Лінійні структури даних
- Не лінійні структури даних

Лінійні структури даних складається з елементів, розташованих у послідовному порядку і кожен елемент з'єднується один за одним.

Не лінійні структури даних не мають певної послідовності з'єднання всіх своїх елементів, і кожен елемент може мати кілька шляхів для приєднання до інших елементів. Такі структури даних нелегко реалізувати, але вони більш ефективні у використанні пам'яті. Деякі приклади не лінійних структур даних – дерево, граф.

Також лінійні структури даних поділяються на:

- Статичні структури даних
- Динамічні структури даних

У статичній структурі даних розмір структури фіксований. Вміст структури даних можна змінювати, але без зміни виділеного для неї простору пам'яті. Найпоширенішим прикладом статичної структури даних є масив.

У динамічній структурі даних розмір структури не фіксований і може бути змінений під час операцій, що виконуються над нею. Динамічні структури даних призначені для полегшення зміни структур даних під час виконання. Основними динамічними структурами даних є стек, черга та список [13].

1.2.2 Примітивні структури даних

До примітивних структур даних належать такі типи даних як: byte, short, int, long, char, boolean, float, double (таблиця 1.2). Термін примітивні вказує на те, що ці типи даних не являються об'єктами, а лише звичайними двійковими значеннями [2].

У Java чітко визначені області дії примітивних типів та діапазон допустимих значень. Програми на Java повинні бути переносимими, точне дотримання цих правил є однією з основних вимог мови. Це означає, що при переході з однієї платформи на іншу не доведеться переписувати код.

Таблиця 1.2. Примітивні типи даних

Тип даних	Діапазон значень	Опис
byte	-128 до 127	8-розрядне ціле число
short	-32768 до 32767	Коротке ціле число
int	-2147483648 до 2147483647	Ціле число
long	-9223372036854775808 до 9223372036854775807	Довге ціле число
char	0 до 65536	Символ
boolean	true або false	Представляє логічні значення true або false
float	-3.4e+38 до 3.4e+38	Числове значення з плаваючою точкою одинарної точності
double	-1.7e+308 до 1.7e+308	Числове значення з плаваючою точкою подвійної точності

1.2.3 Масиви

Одновимірний масив (Array) - статична структура даних зберігання послідовності значень, які стосуються одному типу (рисунок 1.3). Компоненти масиву називаються його елементами. Доступ до елементів у масиві здійснюється за їх індексами. Нумерація індексів розпочинається з нуля [7].

a0	a1	a2	a3	a4	a5	a6	a7	a8	a9
----	----	----	----	----	----	----	----	----	----

Рисунок 1.3 – Одновимірний масив

Двовимірний масив може розглядатися як масив одновимірних масивів (рисунок 1.4). Якщо елементи одновимірного масиву індексуються одним цілим числом, елементи двовимірного масиву індексуються двома цілими числами: перший індекс задає рядок, а другий – стовпець.

a00	a01	a0,2
a10	a11	a12
a20	a21	a22

Рисунок 1.4 – Двовимірний масив

Таким чином масив – це структура даних, яка характеризується:

- Фіксованим набором елементів одного і того ж типу;
- Кожен елемент має унікальний набір значень індексів;
- Кількість індексів визначають розмірність масиву;
- Звернення до елементу масиву виконується по імені масиву і значенням індексів для даного елементу.

1.2.4 Стек

Стек (Stack) – це динамічна структура даних, яка діє за принципом LIFO (Останній прийшов першим вийшов) (рисунок 1.5). Доповнення нових та видалення існуючих елементів проводиться тільки з одного боку, який називається вершиною стеку [6].

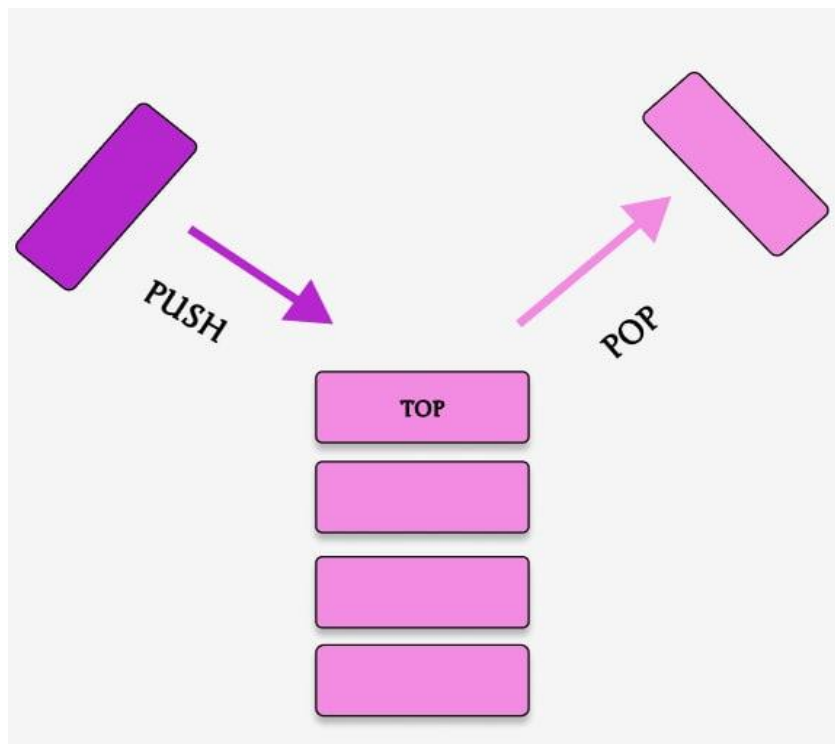


Рисунок 1.5 – Стек

Основні методи стеку є:

push() – занесення елементу до стеку.

pop() – вилучення елемента зі стека.

peek() – повертає вершину стека.

isEmpty() – повертає true якщо стек порожній.

size() – повертає розмір стеку.

1.2.5 Черга

Черга (Queue) - це динамічна структура даних, яка діє за принципом LIFO (останній прийшов, першим вийшов) (рисунок 1.6) [12]. Доповнення елемента відбувається з однієї сторони (з хвоста), а вилучення елемента з іншої (з голови).

Основні методи черги:

enqueue() – додавання елемента до черги.

dequeue() – видалення елемента з черги.

isEmpty() – повертає true якщо черга порожня.

size() – повертає розмір черги.

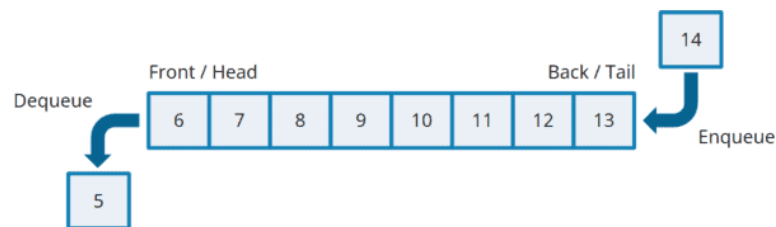


Рисунок 1.6 – Черга

Розширеною версією черги – є двобічна черга (Deque) (рисунок 1.7) [12]. Головною відмінністю двобічної черги від звичайної черги – є те, що доповнення і вилучення елементів може відбуватися з обох сторін.

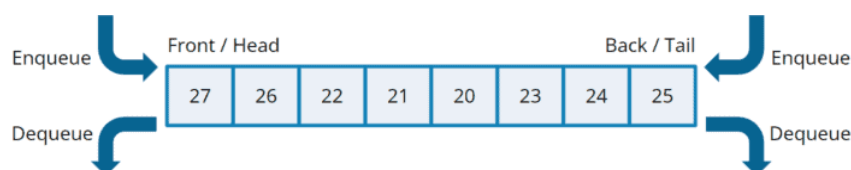


Рисунок 1.7 – Двобічна черга

1.2.6 Зв'язні списки

Зв'язний список – це структура даних, яка складається з вузлів (Node). Списки досить ефективні щодо операцій додавання або видалення елементу в довільному місці списку, виконуючи їх за постійний час. Списки використовуються замість масивів для зберігання й опрацювання однотипних даних, кількість яких заздалегідь є невідома й може змінюватися у процесі роботи. У списках просто організувати процеси видалення елементу чи його вставку на довільне місце [14].

Вузли (Node) є фундаментальними будівельними блоками багатьох структур даних. Вони є основою для зв'язаних списків, стеків, черг, дерев тощо.

Окремий вузол містить поля даних та посилань на інші вузли. Кожна структура даних додає до цих функцій додаткові обмеження або поведінку, щоб створити потрібну структуру [16].

За кількістю полів вузла розрізняють :

- Однозв'язний список (посилання на наступний елемент)
- Двозв'язний список (посилання на наступний і попередній елемент)

За способом зв'язку вузлів розрізняють:

- Лінійні списки (останній елемент вказує на NULL)
- Циклічні списки (останній елемент вказує на перший)

Однозв'язний список – це структура даних, яка складається з двох вузлів, перший – частина даних, друга – посилання на наступний елемент (рисунок 1.8).

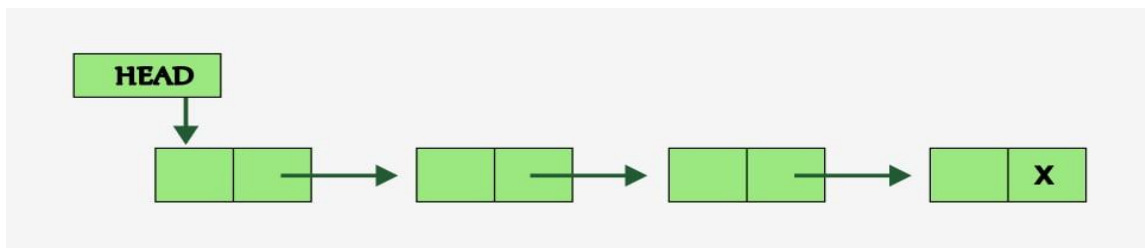


Рисунок 1.8 – Однозв'язний список

В однозв'язному списку доступ можна отримати лише в одному напрямку. Він використовує менше місця в пам'яті та є менш ефективний у порівнянні з двозв'язним списком. Для цієї структури потрібна лише одна змінна вказівника списку, тобто головний покажчик, що вказує на перший вузол. У однозв'язному списку тимчасова складність для вставки та видалення елемента зі списку $O(n)$.

Двозв'язний список – це структура даних, яка складається з трьох вузлів, перший – частина даних, друга – посилання на наступний елемент, третя – посилання на попередній елемент (рисунок 1.9).

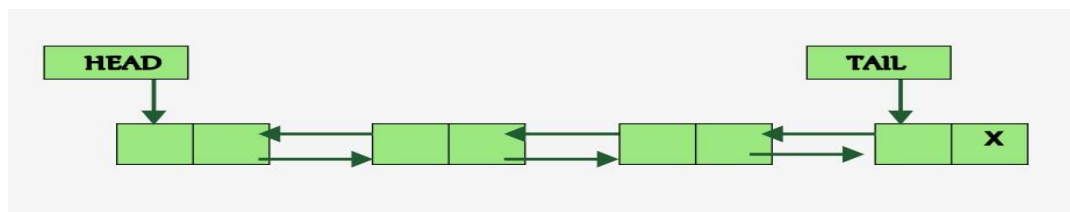


Рисунок 1.9 – Двозв'язний список

До двозв'язаного списку можна отримати доступ в обох напрямках. Він використовує більше місця в пам'яті та є більш ефективним у порівнянні з однозв'язним списком. Для цієї структури потрібні дві змінні вказівника списку, які вказують на перший вузол та останній. У двозв'язному списку тимчасова складність для вставки та видалення елемента дорівнює $O(1)$ [15].

1.2.7 Дерево

Дерево – це нелінійна структура даних, тобто не має певної послідовності з'єднання всіх своїх елементів, що являє собою сукупність елементів і відносин, що утворюють ієрархічну структуру цих елементів. Дерево складається з вузлів, з'єднаних ребрами (рисунок 1.10). Кожен вузол містить значення або дані, і може мати (або не мати) дочірні вузли. Ребра показують зв'язки між вузлами. Початковий вузол дерева називають коренем дерева, йому відповідає нульовий рівень. Листями дерева називають вершини, в які входить одна гілка і не виходить жодної гілки [8].

Кожне дерево має такі властивості:

- Існує вузол, в який не входить ні одна дуга (корінь);
- У кожену вершину, крім кореня, входить одна дуга.

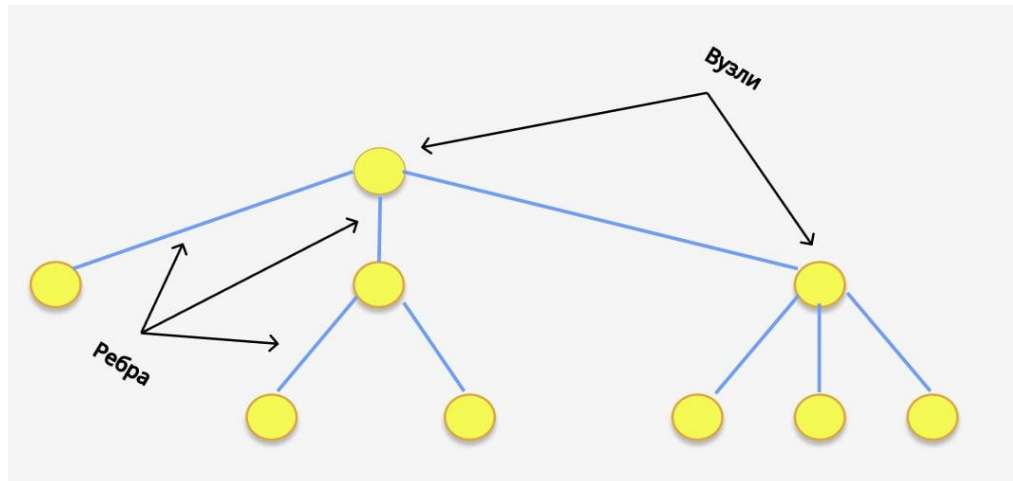


Рисунок 1.10 – Дерево

Бінарне (двійкове) дерево є різновидом дерев (рисунок 1.11) [17].

Двійкове дерево – це структура даних, яка я дуже подібною до звичайного дерева, але є деякі відмінності:

- У кожного вузла не більше двох дітей.
- Будь-яке значення менше значення вузла стає лівою дитиною або дитиною лівої дитини.
- Будь-яке значення більше або дорівнює значенню вузла стає правою дитиною або дитиною правої дитини.

Кожен вузол має ключ і пов'язане значення. Під час пошуку потрібний ключ порівнюється з ключами в BST, і якщо знайдено, витягується пов'язане значення. Основні операції в бінарному дереві пошуку виконуються за час, пропорційний його висоті. Для повного бінарного дерева з n вузлами ці операції виконуються за час $O(\log n)$ у найгіршому випадку [17].

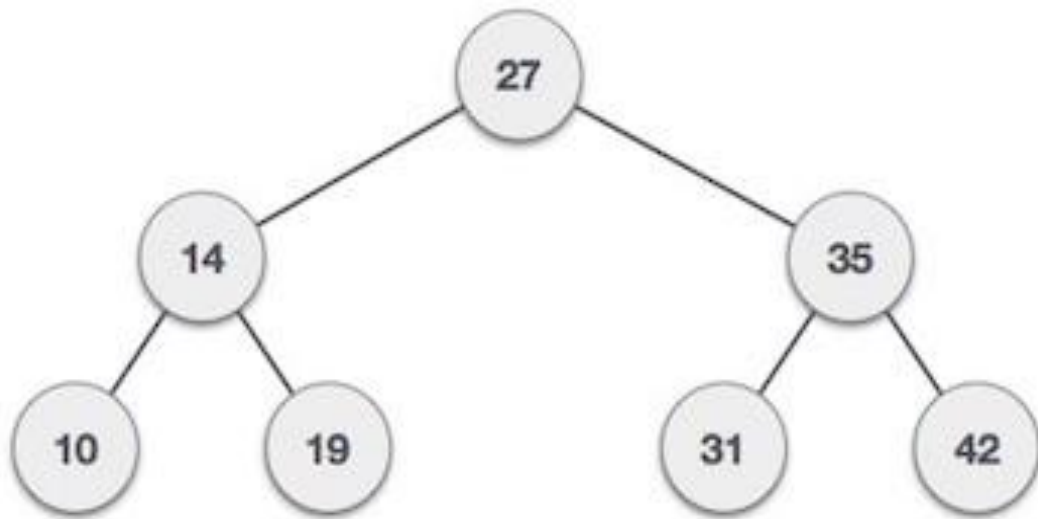


Рисунок 1.11 – Бінарне дерево

Процес доступу до елементів дерева називається проходженням дерева.

Існує три способи проходження дерев (рисунок 1.12):

- Послідовний
- Низхідний
- Висхідний

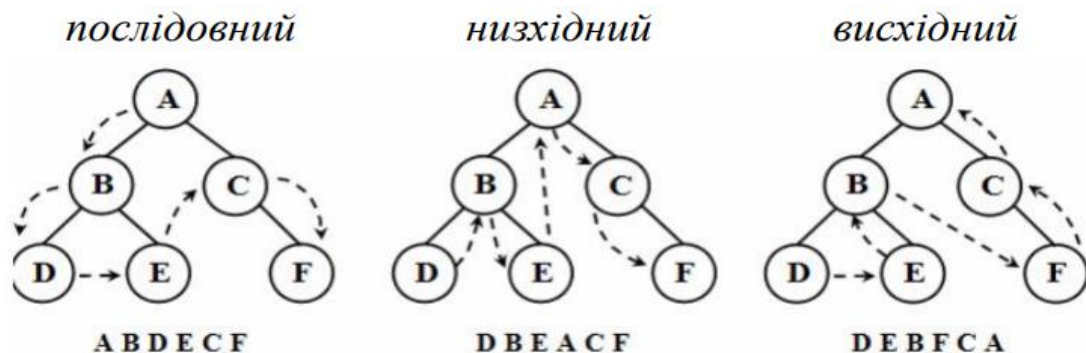


Рисунок 1.12 – Способи проходження дерев

Перевага дерев полягає в тому, що, не дивлячись на складність їх формування, всі операції з ними є дуже простими. Так, скажімо, операції вставки, видалення та пошуку елементів виконуються в тисячі, або навіть у десятки тисяч разів швидше ніж аналогічні операції з невпорядкованим масивом.

1.3. Граф та алгоритми у графові.

1.3.1 Граф

Граф – це нелінійна структура даних, яка зберігає певні дані. Граф складається з вершин (вузлів) і ребер. Кожна вершина і ребро мають зв'язок. Де вершина містить дані, а ребро представляє відношення між ними [18].

Існують такі типи графів:

- Орієнтований граф
- Неорієнтований граф
- Зважений граф
- Незважений граф

Орієнтований граф – граф, ребра якого є однонаправленими, можна рухатися в одному напрямку (рисунок 1.13) [18].

Неорієнтований граф – граф, ребра якого є двонаправленими, можна пройти в будь-якому напрямку (Рисунок 1.13) [18].

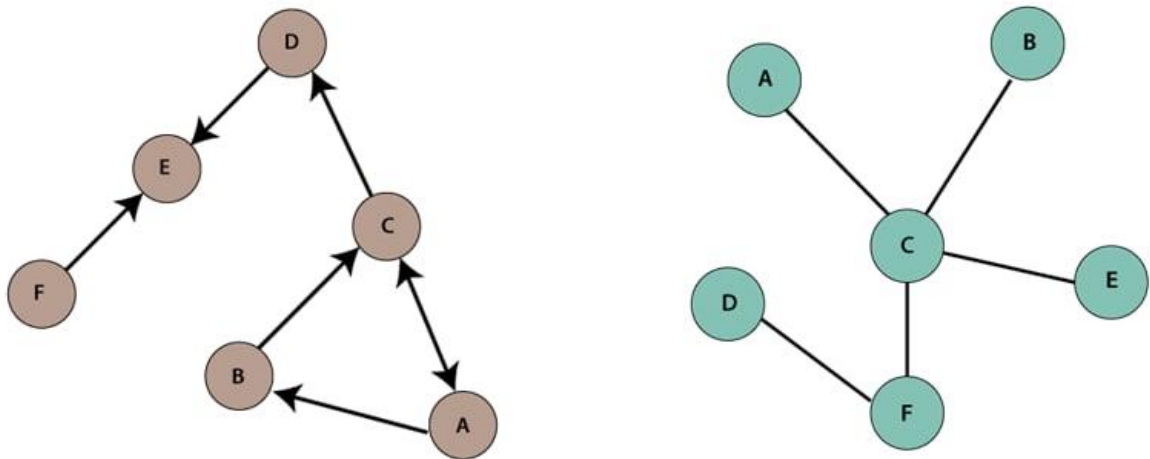


Рисунок 1.13 – Орієнтований та неорієнтований граф

Зважений граф – граф, у якому кожне ребро містить деякі дані (вагу), такі як відстань, вага, висота тощо. Використовується для обчислення вартості переходу від однієї вершини до іншої (рисунок 1.14) [18].

Незважений граф - граф, у якому ребра не пов'язані з жодним значенням (рисунок 1.14) [18].

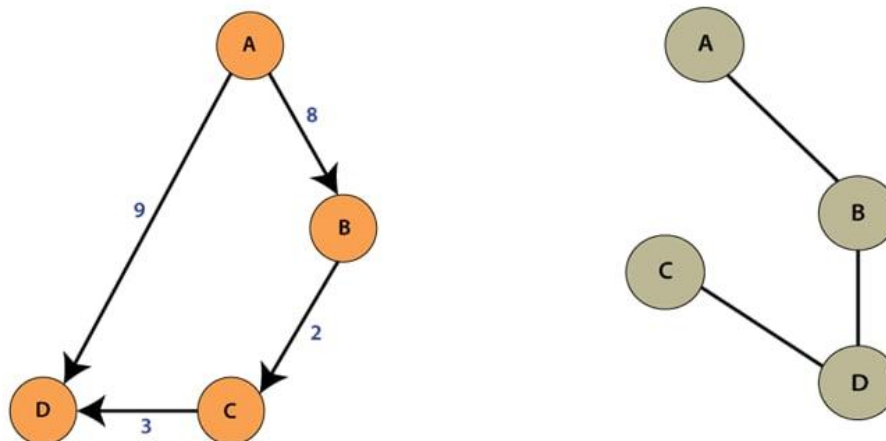


Рисунок 1.14 – Зважений та незважений граф[18]

Маршрутом в графі називається послідовність вершин і ребер, яка має такі властивості:

- Вона починається і закінчується вершиною;
- вершини і ребра в ній чергуються;

Будь-яке ребро цієї послідовності має своїми кінцями дві вершини: що безпосередньо передує йому в цій послідовності і наступну що йде відразу за ним.

Шлях називається замкненим або циклічним, якщо він починається та закінчується у одній і тій же вершині [4].

Існує два основних варіанта подання графів (рисунок 1.15):

- Матриця суміжності
- Список суміжності

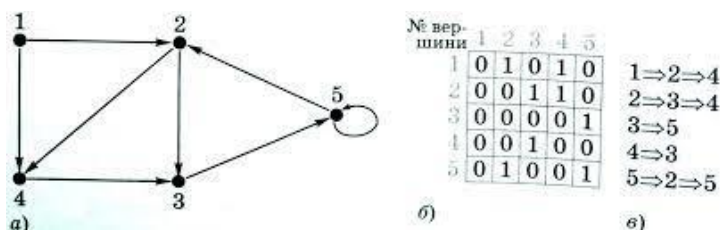


Рисунок 1.15 – Варіанти представлення графа

Між графами та деревами є прямий зв'язок, що полягає у тому, що кожне дерево є орієнтованим зв'язним графом, якщо передбачається рух лише від вузла до синів, або неорієнтованим, якщо додатково допускається рух від синів до батьків [1].

1.3.2 Пошук в ширину(BFS) та в глибину(DFS)

Обхід графа – це проходження його вершин за певними властивостями і критеріями. Двома основними методами обходу графа є пошук в ширину та пошук у глибину. Незважаючи на те, що обидва алгоритми використовуються для обходу графа, вони мають деякі відмінності.

Пошук у ширину (англ. Breadth-first search, BFS) — алгоритм для обходу графа, у якому застосовується стратегія послідовного перегляду окремих рівнів графа, починаючи з заданого вузла (рисунок 1.16) [4].

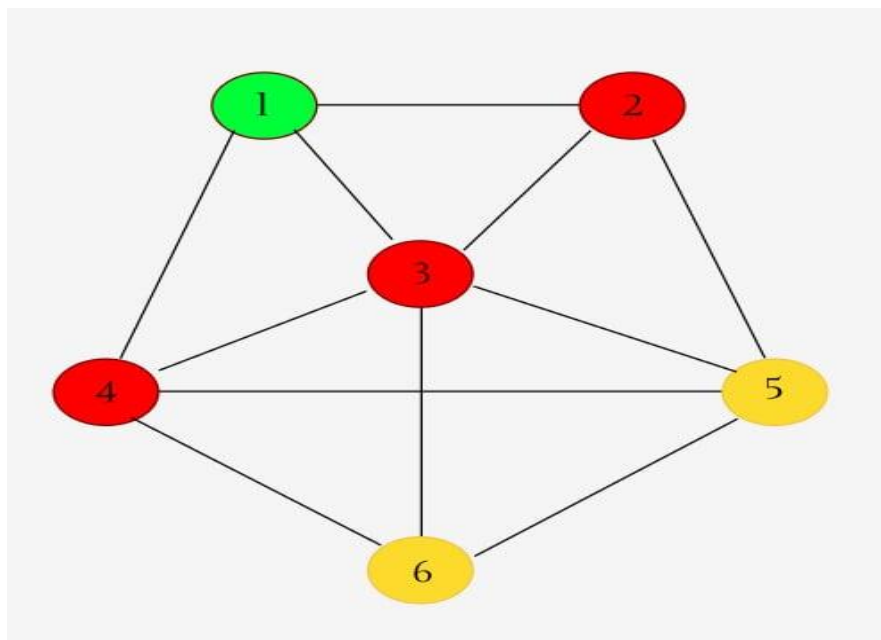


Рисунок 1.16 – Обхід графа у ширину

Незважаючи на те, що BFS може здаватися повільнішим, насправді він швидший, оскільки при роботі з великими графами виявляється, що DFS витрачає багато часу на прямування по шляхах, які зрештою виявляються помилковими.

BFS часто використовується для пошуку найкоротшого шляху між двома вершинами.

Алгоритм в глибину (англ. Depth-first search, DFS) — алгоритм для обходу графа, у якому застосовується стратегія, яка полягає в тому, щоб йти «вглиб» графа наскільки це можливо, а досягнувши глухого кута, повертатися до найближчої вершини, яка має суміжні раніше не відвідані вершини (рисунок 1.17). Вибір початкової вершини під час обходу графа в глибину є важливим з огляду на те, чи стоїть задача обійти всі вершини графа [4].

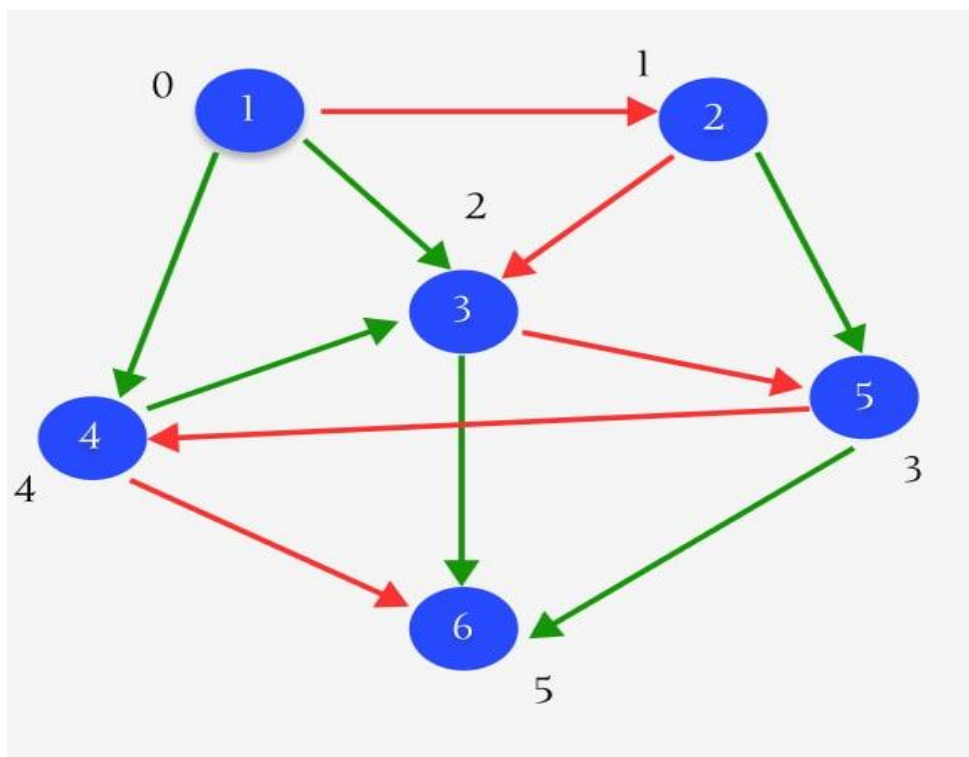


Рисунок 1.17 – Обхід графа у глибину

Алгоритмічна складність пошуку в ширину та в глибину залежить від того, як реалізований граф. У випадку реалізації графа через матрицю суміжності алгоритмічна складність пошуку в ширину $O(n^2)$ якщо граф має n вершин. Тоді, як для у випадку реалізації графу через список суміжності, його алгоритмічна складність $O(n + m)$ у найгіршому випадку, де n – кількість вершин у графові, а m – кількість ребер у графі.

1.3.3 Алгоритм Дейкстрі

Алгоритм Дейкстрі – це алгоритм, який дозволяє знайти найкоротший шлях між будь-якими двома вершинами графа. алгоритм є дієвим лише на зважених графах [19].

На кожній ітерації алгоритм знаходить вузол, який є найближчим до поточного вузла. Потім він досліджує найближчий вузол і повторює наведений вище крок (рисунок 1.18).

Алгоритм вибирає найбільш оптимальний шлях на кожній ітерації, не турбуючись про те, що лежить далі, тому він класифікується як жадібний алгоритм.

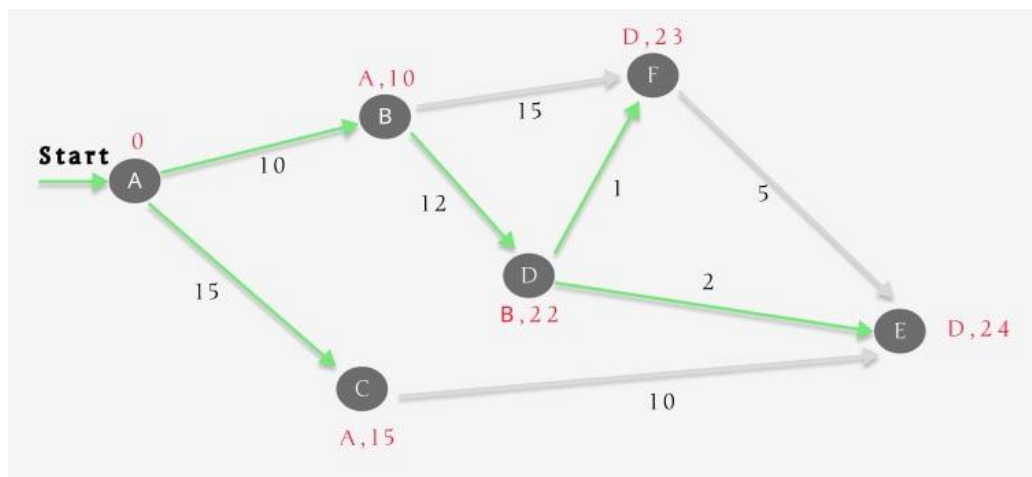


Рисунок 1.18 – Найкоротший шлях до вершин

Основним недоліком алгоритму Дейкстрі є те, що він не може обробляти негативні ваги ребра. Це відбувається тому, що алгоритм Дейкстрі використовує жадібний підхід. Щоразу, коли з масиву відстаней вибирається найближчий вузол, позначається відстань цього вузла як остаточну. Ніколи не буде оновлюватись відстань цього вузла. Це чудово працює для додатних ваг, але може дати непослідовні рішення для негативних ваг ребер [3].

Для кожної ітерації обчислюється відстань від найближчого вузла до всіх інших вузлів. Це призводить до часової складності $O(n^2)$, де n – кількість вершин.

2. РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТУ

2.1. DoublyLinkedList

Клас `DoublyLinkedList` є реалізацією двозв'язного списку. Він приймає параметризований тип даних `E`. Клас імплементує інтерфейс `Iterable<E>`, для подальшого використання циклу `foreach` для елементів екземпляру `DoublyLinkedList`.

У класі `DoublyLinkedList` також присутній клас `Node`, який приймає параметризований тип даних `E`. Він являє собою елемент, який містить дані типу `E` із назвою змінної `elem`, та посилання на наступну ноду `next`, та попередню ноду `prev`. У цьому класі створенні гетери та сетери для кожної змінної (`E elem`, `Node<E> prev`, `Node<E> next`).

Створюються змінні `Node<E> firstNode`, `Node<E> lastNode`, `int size`. У конструкторі класу `DoublyLinkedList` відбувається ініціалізація змінних.

У цьому класі присутні такі методи як:

`size()` – повертає кількість елементів у списку

`isEmpty()` – повертає `true` якщо список порожній, і `false` якщо у списку присутній принаймні один елемент.

`addFirst()` – метод, який добавляє елемент на початок списку.

`addLast()` – метод, який добавляє елемент у кінець списку.

`removeFirst()` – метод, який видаляє перший елемент у списку. Якщо список порожній, викликається виключення `NoSuchElementException`.

`removeLast()` – метод, який видаляє останній елемент у списку. Якщо список порожній, то викликається виключення `NoSuchElementException`.

`remove()` – метод, який приймає у параметр числове значення типу `int`, яке слугує індексом, який потрібно видалити у списку. Якщо передане число у параметр є меншим за нуль, або більше рівним за розмір списку, викликається виключення `NoSuchElementException`.

`clear()` – метод, який очищає список.

`get()` – метод, який приймає у параметр числове значення типу `int`, яке слугує індексом, і повертає елемент списку за цим індексом. Якщо передане число у параметр є меншим за нуль, або більше рівним за розмір списку, викликається виключення `NoSuchElementException`.

`set()` – метод, який у параметр приймає числове значення типу `int`, яке слугує індексом, та значення параметризованого типу `E`, яке слугує елементом. Метод знаходить елемент за заданим індексом, та змінює його на значення, яке було передано у параметрі. Якщо передане число у параметр є меншим за нуль, або більше рівним за розмір списку, викликається виключення `NoSuchElementException`.

`indexOf()` – метод, який у параметр приймає значення параметризованого типу `E`, і у випадку, якщо елемент присутній у списку, то повертає його індекс. У протилежному випадку, викликається виключення `NoSuchElementException`.

`toString()` – перевизначений метод, який призначений для коректного виведення списку.

2.2. Queue

Клас `Queue` є реалізацією черги. Він приймає параметризований тип даних `E`. Клас імплементує інтерфейс `Iterable<E>`, для подальшого використання циклу `foreach` для елементів екземпляру `Queue`.

У цьому класі присутній вкладений клас `Node`, який приймає параметризований тип даних `E`. Він являє собою елемент, який містить дані типу `E` із назвою змінної `elem`, та посилання на наступну ноду `next`. Додатково створено гетери та сетери.

Створюються змінні `Node<E> firstNode`, `Node<E> lastNode`, `int size`. У конструкторі класу `Queue` відбувається ініціалізація змінних.

У цьому класі присутні такі методи як:

`enqueue()` – це метод, який приймає у параметр значення параметризованого типу `E`, та добавляє його на початок до стеку.

`dequeue()` – це метод, який вилучає останнє значення зі стеку. У випадку, якщо стек порожній, викликається виключення `NoSuchElementException`

`isEmpty()` – повертає `true` якщо черга порожня, і `false` якщо у черзі присутній принаймні один елемент.

`Size()` – повертає кількість елементів у черзі.

`toString()` – перевизначений метод, який призначений для коректного виведення черги.

2.3. BST

Клас `BST` є реалізацію бінарного дерева. Клас `BST` приймає два узагальнених класи: `Key` та `Value`. Клас `Key` є наслідником класу `Comparable<>`. Це потрібно для можливості порівнювати екземпляри класу `Key` та знаходження певного елемента в дереві. Клас `Value` – це клас, екземпляри якого будуть зберігатися в кожному вузлі дерева, і містити у собі якусь інформацію.

У цьому класі присутній вкладений клас `Node`, який являє собою реалізацію вузла бінарного дерева. Даний клас містить екземпляр класу `Key` – `key`, екземпляр класу `Value` – `value`, та два екземпляри класу `Node`: `left` та `right`, та змінну `int size`. `key` – це ключ, по якому даний вузол перевіряється. `Value` – це інформація, значення, яке зберігається у даному вузлі. `Left` – посилання на лівий вузол. `Right` – посилання на правий вузол. Конструктор класу `Node` приймає екземпляр класу `Key` – `key`, та екземпляр класу `Value` – `value`, та екземпляр класу `int` – `size`. В тілі конструктора відбувається ініціалізація відповідних змінних. Клас `BST` містить екземпляр класу `Node` – `root`, який являє собою реалізацію кореня даного бінарного дерева. Конструктора клас `BST` не має.

В основному, усі методи у цьому класі побудовані на рекурсії, тому більшість методів створенні за допомогою перегрузки методів.

Даний клас містить такі методи:

`isEmpty()` – повертає `true` якщо бінарне дерево порожнє, і `false` якщо у ньому присутній принаймні один елемент.

`size()` – який повертає кількість елементів у дереві.

`put()` – це метод, який у параметр приймає ключ типу `Key`, та значення типу `Value`. Метод добавляє у дерево ключ та значення. Також у випадку якщо ключ у дереві уже існує, то метод змінює значення вузла. У випадку, якщо ключ переданий у параметр рівний `null`, викликається виключення `NullPointerException`. А якщо передане значення дорівнює `null`, видаляється вузол, із значенням ключа.

`select()` – це метод, який у параметр приймає числове значення типу `int`, яке слугує рангом вузла, та повертає вузол який містить ключ ранга. У випадку, якщо задане числове значення у параметрі менше за нуль, або більше рівне за розмір дерева, викликається виключення

`get()` – це метод, який у параметр приймає ключ типу `Key`, та повертає значення вузла за заданим ключем. У випадку якщо заданого ключа не існує у дереві, то повертає `null`.

`min()` – це метод, який повертає ключ мінімального вузла дерева. У випадку якщо дерево порожнє, викликається виключення `NoSuchElementException`.

`max()` – це метод, який повертає ключ максимального вузла дерева. У випадку якщо дерево порожнє.

`deleteMin()` – це метод, який видаляє мінімальний вузол дерева. У випадку якщо дерево порожнє, викликається виключення `NoSuchElementException`.

`deleteMax()` – це метод, який видаляє максимальний вузол дерева. У випадку якщо дерево порожнє.

`delete()` – це метод, який видаляє вузол дерева, за заданим у параметрі ключем. У випадку, якщо ключ рівний `null`, викликається виключення `NullPointerException`.

`keys()` – це метод, який повертає `Iterable<Key>`, тим самим виводить усі ключі дерева.

2.4. Vertex

Клас `Vertex` є реалізацією вершини графу. Клас містить у собі назву вершини, яка є параметризованим типом `E` із назвою змінної `object`, та список

DoublyLinkedList<Vertex<E>> із назвою vertices, яка містить у собі усі вершини, з'єднані з даною вершиною.

Конструктор класу приймає параметризований тип даних E із назвою object та ініціалізує список vertices.

Створені гетери і сетери для змінних object та vertices, які дозволяють звертатися або змінювати об'єкти.

Даний клас містить такі методи:

connectVertex() – це метод, який у параметр приймає значення типу Vertex<E>, та додає його у список vertices, тим самим, показуючи, що ці вершини тепер з'єднані.

disconnectVertex() – це метод, який у параметр приймає значення типу Vertex<E>, та вилючає його зі списку vertices, тим самим, показуючи, що ці вершини тепер роз'єднані.

toString() – перевизначений метод, який призначений для коректного виведення елемента цього класу.

2.5. Edge

Клас Edge є реалізацією ребра графу. У цьому класі є два об'єкта класу Vertex<E>, а саме ver1 і ver2, та змінна weight типу int, яка відповідає за вагу ребра. Конструктор класу приймає два значення параметризованого типу E, та одне числове значення типу int. У тілі конструктора ініціалізуються вершини ver1 та ver2 а також ініціалізується змінна weight.

Створені гетери для змінних ver1, ver2 та weight, які дозволяють звертатися об'єкта. Також перевизначено метод toString() для коректного виведення елемента цього класу.

2.6. Graph

Клас Graph є реалізацією зваженого графу. Він містить масив класу Vertex – vertices, та список класу Edge – edges, та список класу E – data. Конструктор класу Graph приймає список вершин data. У тілі конструктора ініціалізується

масив `vertices` з розміром, рівним `info. Size()` , а потім заповнюється новими елементами класу `Vertex`, в конструктор яких передається елементи зі списку `data`.

Створені гетери для змінних `vertices`, `edges` та `data`, які дозволяють звертатися об'єкта.

Даний клас містить такі методи:

`sizeVertex()` – це метод, який повертає кількість вершин у графі, тобто він повертає розмір масиву `vertices`.

`sizeEdges()` – це метод, який повертає кількість ребер у графі, тобто він повертає розмір списку `edges`.

`addEdge()` – це метод, який приймає у параметр назви двох вершин, та вагу ребра, та створює ребро з відповідними значеннями, та добавляє ребро до списку `edges`, та з'єднує першу вершину з другою.

`deleteEdge()` – це метод, який приймає у параметр назви двох вершин, та видаляє ребро з відповідними значеннями, та роз'єднує першу вершину з другою.

`getVertex()` – це метод, який у параметр приймає значення типу `E`, яка відіграє роль вершини, та у випадку якщо вершину з заданою назвою знайдено, повертає екземпляр класу `Vertex<E>`, у протилежному викликає виключення `NullPointerException`.

`getEdge()` – це метод, який у параметр приймає два значення типу `E`, та у випадку якщо відповідне ребро з заданими вершинами знайдено, повертає екземпляр класу `Vertex<E>`, у протилежному викликає виключення `NullPointerException`.

`edgeSubsist()` – метод, який приймає у параметр два значення типу `E`, і у випадку якщо знайдене ребро з заданими вершинами, повертає `true`, у протилежному випадку повертає `false`.

`getDistance()` – метод, який приймає у параметр два значення типу `E`, і у випадку, якщо існує ребро з заданими вершинами, повертає його вагу, у протилежному випадку, повертає число `1000000`, яке умовно є нескінченністю.

`toString()` – перевизначений метод, який призначений для коректного виведення графу.

2.7. BreadthFirstSearch

Клас `BreadthFirstSearch` є реалізацією обходу в ширину у графові. У класі створені масив `marked` типу `boolean`, для помітки пройдених вершин, та список `result` типу `DoublyLinkedList<Vertex<E>>`, у якому зберігається прохід елементів в ширину. Конструктор класу приймає змінну `graph` типу `Graph` і ініціалізує усі змінні.

Даний клас містить такі методи:

`bfs()` – метод, який у параметр приймає значення `item` типу `E`, і проходить граф у ширину з цієї вершини. У цьому методі акцент робиться на вершинах графа. Спочатку береться вершина із заданою назвою з параметру `item`, і відмічається як відвідана, та добавляється у список `result`. Вершини, суміжні з відвіданою вершиною, потім відвідуються, зберігаються у черзі послідовно, та добавляються у список `result`. Так само збережені вершини потім обробляються одна за одною, та їх сусідні вершини відвідуються та добавляються у список `result`. Метод закінчує свою роботу, коли усі вершини будуть відвідані, та масив `marked` складається лише з значень `true`.

`printRes()` – метод, який виводить порядок вершин, після обходу в ширину.

2.8. DepthFirstSearch

Клас `DepthFirstSearch` є реалізацією обходу в глибину у графові. У класі створені масив `marked` типу `boolean`, для помітки пройдених вершин, та список `result` типу `DoublyLinkedList<Vertex<E>>`, у якому зберігається прохід елементів в глибину. Конструктор класу приймає змінну `graph` типу `Graph` і ініціалізує усі змінні.

Даний клас містить такі методи:

`dfs()` – метод, який у параметр приймає значення `item` типу `E`, і проходить граф у глибину з цієї вершини. Алгоритм побудований на рекурсії, тому пошук у глибину починається з початкової вершини переданої у параметр, вона відмічається як відвідана і додається у список `result`, тоді за допомогою циклу переглядаються суміжні вершини з відвіданою вершиною, і якщо вона не

відмічена, викликається метод `dfs()`, тим самим створюючи рекурсію. Метод закінчує свою роботу, коли усі вершини будуть відвідані, та масив `marked` складається лише з значень `true`.

`printRes()` – метод, який виводить порядок вершин, після обходу в глибину.

2.9. Dijkstra

Клас `Dijkstra` є реалізацією алгоритму Дейкстри у графові. У класі створено цілочисельний масив `D`, у якому будуть зберігатися найкоротші шляхи, від початкової вершини до усіх інших. Також містить масив екземплярів класу `Vertex` – `vertices`, та список `tarryingVertices` типу `DoublyLinkedList<Integer>`, який зберігає номери вершин, найближчі шляхи до яких залишилось обрахувати. Конструктор класу приймає змінну `graph` типу `Graph` і ініціалізує усі змінні.

Даний клас містить такі методи:

`dijkstra()` – метод, який у параметр приймає початковий елемент, маршрут з якого розпочинається. Алгоритм відстежує відому на даний момент найкоротшу відстань від вихідного вузла до кожного вузла і оновлює ці значення, якщо знаходить коротший шлях. Як тільки алгоритм знайшов найкоротший шлях між вихідним вузлом та іншим вузлом, вузол позначається як «відвіданий» і додається до шляху. Процес продовжується до тих пір, поки всі вузли у графі не будуть додані до шляху. Таким чином, маємо шлях, який з'єднує вихідний вузол з усіма іншими вузлами за найкоротшим шляхом для досягнення кожного вузла.

`dijkstraTo()` – метод, який у параметр приймає кінцевий елемент, відстань до якого елемента шукається, та виводить відповідну відстань.

`changeData()` – метод, який у параметр приймає значення типу `E` із назвою `elem`, і зсуває список вершин `data`, таким чином, щоб елемент `elem`, з якого шукається відстань був на першому місці, наступний на другому і т.д.

`saveData()` – метод, який дублює список вершин `data`, та повертає його, тим самим зберігаючи початковий список.

`searchDist()` – метод, який приймає у параметр початкову точку, та кінцеву, і за допомогою циклів шукає маршрут, і повертає список вершин, через які проходить маршрут.

`dijkstraRes()` – метод, який приймає у параметр початкову точку, та кінцеву, і відповідно до цих точок знаходить найкоротшу відстань за допомогою методів `dijkstra()` та `dijkstraTo()`, і виводить його, та знаходить маршрут за допомогою методу `searchDist()` та виводить його.

2.10. ReadInfoForFile

Клас `ReadInfoForFile` відповідає за зчитування даних з файлу csv. Фото файлу представлено в додатку 2.

Даний клас містить такі методи:

`fillingBST()` – метод, який створює об'єкт класу `BST <String, Integer>`, і відповідно заповнює його ключем, тобто назвою міста, відміченою жовтим кольором на фото, та значенням, вагу яку потрібно вивантажити у місті, відміченим червоним кольором на фото.

`readCities()` – метод, який створює список типу `DoublyLinkedList<String>`, та заповнює його назвами міст, відміченим жовтим кольором на фото.

`fillingGraph()` – метод, в параметр якого передається `graph` типу `Graph`, та заповнюємо його ребрами відповідно з файлу. Беремо початкову вершину, відмічену жовтим кольором, та з'єднуємо з першою вершиною відміченою фіолетовим кольором, та встановлюємо вагу ребра першим значенням, відміченим зеленим кольором. Тоді з'єднуємо вершину відмічену жовтим кольором, та відповідно з'єднуємо з другою вершиною відміченою фіолетовим кольором, та встановлюємо вагу ребру другим значенням, відміченим зеленим кольором. Тоді переходимо на наступну початкову вершину, і відповідно пророблюємо ті самі дії, для заповнення графа.

2.11. CarTask

Клас `CarTask` яка створює клас для машини, яка їздить по містах, і вивантажує вантаж, у відповідних містах, та по відповідному маршруту.

У конструктор класу передаються такі змінні, як: `float massCar` – що відповідає за масу автомобіля, `float maxLoad` – що відповідає за максимальну масу вантажу, `float fuelDiesel` – що відповідає за максимальний об’єм палива, `Graph<E>` `graph` – що відповідає за граф, по якому рухається машина, `BST<String, Integer>` `bst` – що відповідає за інформацію вивантаження вантажу у місті. Створюється список `listSum` типу `DoublyLinkedList<Float>`, у якому будуть зберігатися кількість витраченого палива, за пройдений проміжок маршруту. Конструктор ініціалізує усі змінні.

Даний клас містить такі методи:

`info()` – це метод, який виводить інформацію про автомобіль, його назву, масу автомобіля, допустиму масу вантажу, та обсяг дизеля на 100 кілометрів.

`printBaggage()` – метод, який виводить дані з дерева `bst`, тобто виводить назву міста, та масу яку потрібно вивантажити.

`getDisel()` – метод, який у параметр приймає кількість пройдених кілометрів, та масу яка вивантажиться наприкінці маршруту, та в результаті знаходить кількість витраченого палива, та добавляє до списку `listSum`.

`getDisel()` – перевантажений метод, який у параметр приймає список вершин, по якому автомобіль буде маршрут, і визначає обсяг витраченого палива за весь маршрут, та виводить його.

`sumDisel()` – метод, який сумує список `listSum`, та повертає загальний обсяг витраченого палива.

`resetDiesel()` – метод, який очищає список `listSum`, що дозволяє проходити маршрут при повному навантаженні.

2.12. Main

Клас `Main` містить головний метод `main` у якому реалізуються усі структури та алгоритми. Створюються об’єкти класу `ReadInfoForFile`, `DoublyLinkedList`, `BST`, `Graph`, `BreadthFirstSearch`, `DepthFirstSearch`, `Dijkstra` та `CarTask`. Заповнюються необхідні структури, виводиться їх зміст, виконуються різні алгоритми, та також виводяться результати.

3. ПРИКЛАД РОБОТИ З ПРОДУКТОМ

Під час виконання цього завдання побудовано граф, на основі карти заходу України (рисунок 3.1). Усі відстані між містами взяті на офіційних сторінках GOOGLE MAPS і є цілком правильними у реальному часі.

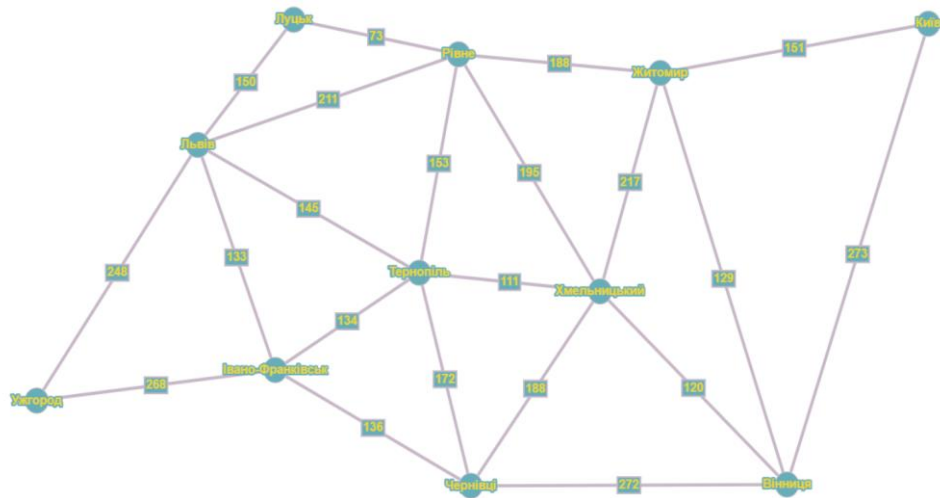


Рисунок 3.1 – Графічне зображення графа

Після заповнення графа даними з файлу, виводиться граф у консоль (рисунок 3.2).

```

Граф міст заходу України
Lutsk --> (Lviv, 150 км); (Rivne, 73 км);
Lviv --> (Uzhhorod, 248 км); (Ivano-Frankivsk, 133 км); (Ternopil, 145 км); (Rivne, 211 км); (Lutsk, 150 км);
Rivne --> (Lutsk, 73 км); (Lviv, 211 км); (Ternopil, 153 км); (Khmel'nitsky, 195 км); (Zhytomyr, 188 км);
Uzhhorod --> (Lviv, 248 км); (Ivano-Frankivsk, 268 км);
Ivano-Frankivsk --> (Uzhhorod, 268 км); (Lviv, 133 км); (Ternopil, 134 км); (Chernivtsi, 136 км);
Ternopil --> (Lviv, 145 км); (Rivne, 153 км); (Khmel'nitsky, 111 км); (Chernivtsi, 172 км); (Ivano-Frankivsk, 134 км);
Zhytomyr --> (Rivne, 188 км); (Khmel'nitsky, 217 км); (Vinnytsia, 129 км); (Kyiv, 151 км);
Khmel'nitsky --> (Ternopil, 111 км); (Rivne, 195 км); (Zhytomyr, 217 км); (Vinnytsia, 120 км); (Chernivtsi, 188 км);
Chernivtsi --> (Ivano-Frankivsk, 136 км); (Ternopil, 172 км); (Khmel'nitsky, 188 км); (Vinnytsia, 272 км);
Kyiv --> (Zhytomyr, 151 км); (Vinnytsia, 273 км);
Vinnytsia --> (Chernivtsi, 272 км); (Khmel'nitsky, 120 км); (Zhytomyr, 129 км); (Kyiv, 273 км);

```

Рисунок 3.2 – Виведення графа у консолі

Створюються об'єкти класу `BreadthFirstSearch` та `DepthFirstSearch`, та виводиться обхід в ширину з вершини Луцьк, та обхід в глибину з вершини Київ (рисунок 3.3).

```
BFS з вершини Lutsk : Lutsk, Lviv, Rivne, Uzhhorod, Ivano-Frankivsk, Ternopil, Khmeltsky, Zhytomyr, Chernivtsi, Vinnytsia, Kyiv,
DFS з вершини Kyiv : Kyiv, Zhytomyr, Rivne, Lutsk, Lviv, Uzhhorod, Ivano-Frankivsk, Ternopil, Khmeltsky, Vinnytsia, Chernivtsi,
```

Рисунок 3.3 – Виведення алгоритмів BFS та DFS у консолі

Створюється об'єкт класу Dijkstra та виводиться алгоритм Дейкстри з вершини Луцьк до вершини Київ. Видаляється ребро Луцьк → Рівне та знову виводиться алгоритм Дейкстри з вершини Луцьк до вершини Київ (рисунок 3.4). В результаті бачимо, що маршрут перебудовується.

```
Найкоротший шлях від Lutsk до Kyiv
Lutsk --> Kyiv : 412 км
[Lutsk, Rivne, Zhytomyr, Kyiv]

Видаємо ребро Lutsk --> Rivne

Найкоротший шлях від Lutsk до Kyiv
Lutsk --> Kyiv : 700 км
[Lutsk, Lviv, Rivne, Zhytomyr, Kyiv]
```

Рисунок 3.4 – Виведення алгоритмів Дейкстри у консолі

Створюється об'єкт класу CarTask, виводиться інформація про автомобіль, інформація про вагу вантажу і місто у якому його потрібно та маршрут по якому поїде авто та обсяги витраченого палива (рисунок 3.5).

```
HYUNDAI HD78
Маса автомобіля: 7500.0 кг
Допустима маса вантажу: 5000.0 кг
Обсяг дизелю на 100 км при повному завантаженні: 18.0 л

Вантаж:
Chernivtsi --> 600 кг; Ivano-Frankivsk --> 900 кг; Khmeltsky --> 1000 кг;
Kyiv --> 300 кг; Lutsk --> 200 кг; Lviv --> 400 кг;
Rivne --> 500 кг; Ternopil --> 800 кг; Uzhhorod --> 700 кг;
Vinnytsia --> 300 кг; Zhytomyr --> 400 кг;

Прокладаємо маршрут: Lutsk --> Lviv --> Rivne --> Zhytomyr --> Kyiv
Lutsk --> Lviv : 150,00 км пройдено , 400,00 кг вивантажено , 27,00 л витрачено
Lviv --> Rivne : 211,00 км пройдено , 500,00 кг вивантажено , 36,76 л витрачено
Rivne --> Zhytomyr : 188,00 км пройдено , 400,00 кг вивантажено , 31,40 л витрачено
Zhytomyr --> Kyiv : 151,00 км пройдено , 300,00 кг вивантажено , 24,35 л витрачено
Всього бензину: 119,52 л
```

Рисунок 3.5 – Виведення результатів завдання згідно варіанту у консолі

ВИСНОВКИ

У курсовій роботі розроблені структури даних, такі як: бінарне дерево, двозв'язний список, черга та граф. Розроблені алгоритми для графа, такі як: пошук в ширину, пошук в глибину та алгоритм Дейкстри. Вивід даних з файлу відбувається з файлу типу .csv. За допомогою файлу, заповнено всі необхідні структури. Розроблено програмний продукт згідно варіанту.

Програмний продукт є доволі актуальним, у теперішній нелегкий військовий час. Він дозволяє вираховувати найкоротший маршрут на основі карти західної частини України, визначати кількість палива, який потрібен на заданий маршрут, та проінформуватися, яку вагу вантажу потрібно вивантажити у кожному місті. У випадку, якщо ділянка дороги перекрита, можна вилучити цю ділянку дороги з графа, і перерахувати маршрут.

Даний програмний продукт можна покращити розширивши граф(додати нові міста, та з'єднати їх ребрами). Також можна розробити методи, які дозволяють у місті вивантажитися вантажом, та одразу ж завантажити деяку кількість вантажу, що дозволить автомобілю не витратити вільне місце.

Код створений на мові програмування Java . Програма розроблена у інтегрованому середовищі розробки програмного забезпечення JetBrains IntelliJ IDEA 2021.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

- 1) Седжвик Р., Уэйн К., Алгоритмы на Java 4-е изд. : Пер. с англ. — М. : ООО “И.Д. Вильямс”, 2013. — 848с.
- 2) Герберт Шилдт , Java: руководство для начинающих, 7-е изд. : Пер. с англ. - СПб. : ООО "Диалектика", 2019. - 816 с.
- 3) Адитья Бхаргава, Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. - СПб.: Питер, 2017. - 288 с.
- 4) Копец Дэвид, К65 Классические задачи Computer Science на языке Java. — СПб.: Питер, 2022 — 288с.
- 5) Седжвик Р., Уэйн К., Computer Science: основы программирования на Java, ООП, алгоритмы и структуры данных. — СПб.: Питер, 2018. — 1072 с.
- 6) Посібник до вивчення курсу «Java-технології та мобільні пристрої». Алгоритми і структури даних. / О. В. Спирінцева, В. В. Герасимов. — ДНУ ім. О. Гончара, 2015. — 98 с. URL: <https://studfile.net/preview/5782860/>
- 7) Курс лекцій з дисципліни «Алгоритми та структури даних» Луцьк : ВНУ імені Лесі Українки, 2021. – 110 с. URL: https://evnuir.vnu.edu.ua/bitstream/123456789/19978/1/kurs_hryshanovych.pdf
- 8) Алгоритми і структури даних, Опорний конспект лекцій. URL: http://dspace.wunu.edu.ua/bitstream/316497/24160/1/fkit_kn_pzs_asd_LEK.pdf
- 9) Лекція № 1, URL: https://d-learn.pnu.edu.ua/data/users/4207/import/lekcija_1.pdf
- 10) Складність алгоритмів. URL: <https://miyklas.com.ua/p/informatica/9-klas/algoritmi-ta-programi-327047/dvovimirnii-masiv-danikh-skladnist-algoritmiv-328117/re-86e1f29c-86b8-4fd8-9e75-c0039f28462b>
- 11) What is a Data Structure. URL: <https://www.mygreatlearning.com/blog/data-structure-tutorial-for-beginners/>
- 12) Difference between Queue and Deque. URL: <https://www.happycoders.eu/algorithms/java-queue-vs-deque/>

13) Static Data Structure vs Dynamic Data Structure. URL:

<https://www.quora.com/What-is-static-and-dynamic-data-structure>

14) 8 известных структур данных, о которых спросят на собеседовании.

URL: <https://proglib.io/p/8-data-structures>

15) Singly Linked List vs Doubly Linked List. URL:

<https://www.javatpoint.com/singly-linked-list-vs-doubly-linked-list>

16) Node: An individual part of a larger data structure. URL:

<https://www.codecademy.com/learn/getting-started-with-data-structures-java/modules/nodes-java/cheatsheet>

17) Data Structure - Binary Search Tree. URL:

https://www.tutorialspoint.com/data_structures_algorithms/binary_search_tree.htm

18) Java Graph. URL: <https://www.javatpoint.com/java-graph>

19) Dijkstra's Shortest Path Algorithm - A Detailed and Visual Introduction.

URL: <https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/>

Додатки

Додаток 1. Код програми

package MyStructures

Queue

```
package MyStructures;

import java.util.Iterator;
import java.util.NoSuchElementException;

public class Queue<E> implements Iterable<E> {

    private Node<E> firstNode;
    private Node<E> lastNode;
    private int size;

    public Queue() {
        lastNode = new Node<>(null, null);
        firstNode = new Node<>(null, null);
        size = 0;
    }

    public void enqueue(E item) {

        Node<E> oldLast = lastNode;
        lastNode = new Node(item, null);
        if (isEmpty()) firstNode = lastNode;
        else oldLast.next = lastNode;

        size++;
    }

    public E dequeue() {
        if (isEmpty()) throw new NoSuchElementException("Queue underflow");
        E item = firstNode.getElem();
        firstNode = firstNode.next;
        size--;
        if (isEmpty()) lastNode = null;
        return item;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public int size() {
        return size;
    }
}
```



```

@Override
public String toString() {

    if (size == 0) return "Queue is clear";
    StringBuilder sb = new StringBuilder();
    sb.append("[");

    Node<E> current = firstNode;

    while (current != null) {
        sb.append(current.getElem());
        current = current.next;
        if (current != null) sb.append(", ");
    }

    sb.append("]");
    return sb.toString();
}

```

```

private class Node<E> {

    private E elem;
    private Node<E> next;

    private Node(E elem, Node<E> next) {
        this.elem = elem;

        this.next = next;
    }

    public E getElem() {
        return elem;
    }

    public void setElem(E elem) {
        this.elem = elem;
    }

    public Node<E> getNext() {
        return next;
    }

    public void setNext(Node<E> next) {
        this.next = next;
    }
}

```

```

    }

    @Override
    public Iterator<E> iterator() {
        return new QueueIterator(firstNode);
    }

    private class QueueIterator implements Iterator<E> {
        private Node<E> current;

        public QueueIterator(Node<E> first) {
            current = first;
        }

        public boolean hasNext() {
            return current != null;
        }

        public void remove() {
            throw new UnsupportedOperationException();
        }

        public E next() {
            if (!hasNext()) throw new NoSuchElementException();
            E item = current.elem;
            current = current.next;
            return item;
        }
    }
}

```

package MyStructures

BST

```

package MyStructures;

import java.util.NoSuchElementException;

public class BST<Key extends Comparable<Key>, Value> {

    private Node root;

    public boolean isEmpty() {
        return size() == 0;
    } //метод для перевірки на пустоту

    public int size() {
        return size(root);
    } //метод для виведення розміру дерева
}

```

```

private int size(Node x) {
    if (x == null) return 0;
    else return x.size;
}

public void put(Key key, Value val) { //метод для добавлення ключа і значення
    if (key == null) throw new NullPointerException("first argument to put() is null");
    if (val == null) {
        delete(key);
        return;
    }
    root = put(root, key, val);
}

private Node put(Node x, Key key, Value value) {
    if (x == null) return new Node(key, value, 1);
    int cmp = key.compareTo(x.key);
    if (cmp < 0) x.left = put(x.left, key, value);
    else if (cmp > 0) x.right = put(x.right, key, value);
    else x.value = value;
    x.size = 1 + size(x.left) + size(x.right);
    return x;
}

public Key select(int k) { //метод для звертання до ключа
    if (k < 0 || k >= size()) throw new IllegalArgumentException("argument to select() is bigger than
size or < 0");
    Node x = select(root, k);
    return x.key;
}

private Node select(Node x, int k) {
    if (x == null) return null;
    int t = size(x.left);
    if (t > k) return select(x.left, k);
    else if (t < k) return select(x.right, k-t-1);
    else return x;
}

public Value get(Key key) { //метод який повертання значення за ключем
    return get(root, key);
}

private Value get(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) return get(x.left, key);
    else if (cmp > 0) return get(x.right, key);
    else return x.value;
}

```

```

public Key min() {//метод для звертання до мінімального ключа
    if (isEmpty()) throw new NoSuchElementException("called min() with empty symbol table");
    return min(root).key;
}

private Node min(Node x) {
    if (x.left == null) return x;
    else return min(x.left);
}

public Key max() {//метод для звертання до максимального ключа
    if (isEmpty()) throw new NoSuchElementException("called max() with empty symbol table");
    return max(root).key;
}

private Node max(Node x) {
    if (x.right == null) return x;
    else return max(x.right);
}

public void deleteMin() {//видалення мінімального елементу
    if (isEmpty()) throw new NoSuchElementException("MyStructures.BST underflow");
    root = deleteMin(root);
}

private Node deleteMin(Node x) {
    if (x.left == null) return x.right;
    x.left = deleteMin(x.left);
    x.size = size(x.left) + size(x.right) + 1;
    return x;
}

public void deleteMax() {//видалення максимального елементу
    if (isEmpty()) throw new NoSuchElementException("MyStructures.BST underflow");
    root = deleteMax(root);
}

private Node deleteMax(Node x) {
    if (x.right == null) return x.left;
    x.right = deleteMax(x.right);
    x.size = size(x.left) + size(x.right) + 1;
    return x;
}

public void delete(Key key) {//метод для виведення ключа
    if (key == null) throw new NullPointerException("argument to delete() is null");
    root = delete(root, key);
}

private Node delete(Node x, Key key) {
    if (x == null) return null;

```

```

int cmp = key.compareTo(x.key);
if (cmp < 0) x.left = delete(x.left, key);
else if (cmp > 0) x.right = delete(x.right, key);
else {
    if (x.right == null) return x.left;
    if (x.left == null) return x.right;
    Node t = x;
    x = min(t.right);
    x.right = deleteMin(t.right);
    x.left = t.left;
}
x.size = size(x.left) + size(x.right) + 1;
return x;
}

public Iterable<Key> keys() { //метод для виведення ключів дерева
    return keys(min(), max());
}

public Iterable<Key> keys(Key lo, Key hi) {
    if (lo == null) throw new NullPointerException("first argument to keys() is null");
    if (hi == null) throw new NullPointerException("second argument to keys() is null");

    Queue<Key> queue = new Queue<Key>();
    keys(root, queue, lo, hi);
    return queue;
}

private void keys(Node x, Queue<Key> queue, Key lo, Key hi) {
    if (x == null) return;
    int cmplo = lo.compareTo(x.key);
    int cmphi = hi.compareTo(x.key);
    if (cmplo < 0) keys(x.left, queue, lo, hi);
    if (cmplo <= 0 && cmphi >= 0) queue.enqueue(x.key);
    if (cmphi > 0) keys(x.right, queue, lo, hi);
}

private class Node {
    private Key key;
    private Value value;
    private Node left, right;
    private int size;

    public Node(Key key, Value value, int size) {
        this.key = key;
        this.value = value;
        this.size = size;
    }
}

```

package MyStructures

DoublyLinkedList

```
package MyStructures;

import java.util.Iterator;
import java.util.NoSuchElementException;

public class DoublyLinkedList<E> implements Iterable<E> {

    private Node<E> firstNode;
    private Node<E> lastNode;
    private int size = 0;

    public DoublyLinkedList() {

        lastNode = new Node<E>(null, firstNode, null);
        firstNode = new Node<E>(null, null, lastNode);
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public void addFirst(E e) {
        Node<E> next = firstNode;
        next.setElem(e);
        firstNode = new Node<>(null, null, next);
        next.setPrev(firstNode);
        size++;
    }

    public void addLast(E e) {
        Node<E> prev = lastNode;
        prev.setElem(e);
        lastNode = new Node<E>(null, prev, null);
        prev.setNext(lastNode);
        size++;
    }

    public void removeFirst() {
        if (isEmpty()) throw new NoSuchElementException("List is clear");
        firstNode = firstNode.getNext();
        size--;
    }
}
```

```

}

public void removeLast() {
    if (isEmpty()) throw new NoSuchElementException("List is clear");
    lastNode = lastNode.getPrev();
    size--;
}

public void remove(int index) {
    if (index < 0 || index >= size) throw new NoSuchElementException("No element index");
    if (index == 0) {
        removeFirst();

    } else if (index == size() - 1) {
        removeLast();

    } else {

        Node<E> temp = firstNode.getNext();
        for (int i = 0; i < index - 1; i++) {
            temp = temp.next;
        }
        Node<E> temp2 = temp.next;
        temp.setNext(temp2.next);
        temp2.next.setPrev(temp);
        temp2.setElem(null);
        size--;

    }

}

public void clear() {
    size = 0;
    lastNode = new Node<E>(null, firstNode, null);
    firstNode = new Node<E>(null, null, lastNode);

}

public E get(int index) {

    if (index < 0 || index >= size) throw new NoSuchElementException("No element index");
    Node<E> temp = firstNode.getNext();
    for (int i = 0; i < index; i++) {
        temp = getNext(temp);
    }
    return temp.getElem();
}

public void set(int index, E value) {
    if (index < 0 || index >= size) throw new NoSuchElementException("No element index");
    Node<E> temp = firstNode.getNext();

```

```

        for (int i = 0; i < index; i++) {
            temp = getNext(temp);
        }
        temp.setElem(value);
    }

    public int indexOf(E elem) {
        Node<E> temp = firstNode.getNext();
        int index = -1;
        for (int i = 0; i < size; i++) {
            if (elem.toString().equals(temp.getElem().toString())) {
                index = i;
                break;
            }
            temp = getNext(temp);
        }

        if (index == -1) throw new NoSuchElementException("No element index");
        return index;
    }

    @Override
    public String toString() { //Перевизначимо метод toString

        if (size == 0) return "List is clear";
        StringBuilder sb = new StringBuilder();
        sb.append("[");

        Node<E> current = firstNode.getNext();

        while (current != null) {

            sb.append(current.getElem());
            if (current.next == lastNode) {
                break;
            }
            current = current.next;
            if (current != null) sb.append(", ");
        }

        sb.append("]");
        return sb.toString();
    }

    @Override
    public Iterator<E> iterator() {
        return new Iterator<E>() {
            int counter = 0;

            @Override
            public boolean hasNext() {
                return counter < size;
            }
        };
    }

```



```

    }

    @Override
    public E next() {
        return get(counter++);
    }
};
}

private Node<E> getNext(Node<E> current) {
    return current.getNext();
}

private class Node<E> {

    private E elem;
    private Node<E> prev;
    private Node<E> next;

    private Node(E elem, Node<E> prev, Node<E> next) {
        this.elem = elem;
        this.prev = prev;
        this.next = next;
    }

    public E getElem() {
        return elem;
    }

    public void setElem(E elem) {
        this.elem = elem;
    }

    public Node<E> getNext() {
        return next;
    }

    public void setNext(Node<E> next) {
        this.next = next;
    }

    public Node<E> getPrev() {
        return prev;
    }

    public void setPrev(Node<E> prev) {
        this.prev = prev;
    }

}
}

```

Package Graph

Vertex

```

package Graph;

import MyStructures.DoublyLinkedList;

public class Vertex<E> {

    private E object;
    private DoublyLinkedList<Vertex<E>> vertices;

    public Vertex(E object) {
        this.object = object;
        vertices = new DoublyLinkedList<>();
    }

    public E getObject() {
        return object;
    }

    public void setObject(E object) {
        this.object = object;
    }

    public DoublyLinkedList<Vertex<E>> getVertices() {
        return vertices;
    }

    public void setVertices(DoublyLinkedList<Vertex<E>> vertices) {
        this.vertices = vertices;
    }

    public void connectVertex(Vertex<E> v) {
        vertices.addLast(v);
    }

    public void disconnectVertex(Vertex<E> v) {
        vertices.remove(this.getVertices().indexOf(v));
    }

    public String toString() {
        return String.valueOf(object);
    }

}

```

Package Graph Edge

```
package Graph;

public class Edge<E> {

    private final Vertex<E> ver1, ver2;
    private int weight;

    public Edge(E one, E two, int weight) {

        ver1 = new Vertex<>(one);
        ver2 = new Vertex<>(two);
        this.weight = weight;
    }

    public Vertex<E> getVer1() {
        return ver1;
    }

    public Vertex<E> getVer2() {
        return ver2;
    }

    public int getWeight() {
        return weight;
    }

    public String toString() {
        return "(" + ver2.getObject() + ", " + weight + " км)";
    }

}
```

Package Graph Graph

```
package Graph;

import MyStructures.DoublyLinkedList;

public class Graph<E> {

    private final Vertex<E>[] vertices;
    private final DoublyLinkedList<Edge<E>> edges;
    private DoublyLinkedList<E> data;

    public Graph(DoublyLinkedList<E> data) {
        this.data = data;
        vertices = new Vertex[data.size()];
    }
}
```

```

    for (int i = 0; i < data.size(); i++) {
        vertices[i] = new Vertex<>(data.get(i));
    }
    edges = new DoublyLinkedList<>();
}

public int sizeVertex() {
    return vertices.length;
}

public int sizeEdges() {
    return edges.size();
}

public void addEdge(E one, E two, int weight) {
    Edge<E> e = new Edge<>(one, two, weight);
    edges.addLast(e);
    vertices[data.indexOf(one)].connectVertex(vertices[data.indexOf(two)]);
}

public void deleteEdge(E one, E two) {
    edges.remove(edges.indexOf(getEdge(one, two)));
    vertices[data.indexOf(one)].disconnectVertex(vertices[data.indexOf(two)]);
}

public Vertex<E> getVertex(E e) {
    for (Vertex<E> v : vertices) {
        if (String.valueOf(v.getObject()).equals(String.valueOf(e)))
            return vertices[data.indexOf(e)];
    }
    throw new NullPointerException("Вершины " + e.toString() + " не существует");
}

public Edge<E> getEdge(E one, E two) {
    for (Edge<E> e : edges) {
        if (one.toString().equals(e.getVer1().toString()) &&
two.toString().equals(e.getVer2().toString())) {
            return e;
        }
    }
    throw new NullPointerException("ребра " + one + " --> " + two + " не существует");
}

public boolean edgeSubsist(E one, E two) {
    for (int i = 0; i < edges.size(); i++) {

```

```

        if (one.toString().equals(edges.get(i).getVer1().toString()) &&
two.toString().equals(edges.get(i).getVer2().toString())) {
            return true;
        }
    }
    return false;
}

public int getDistance(E one, E two) {
    if (edgeSubsist(one, two))
        return getEdge(one, two).getWeight();
    return 1000000;
}

public Vertex<E>[] getVertices() {
    return vertices;
}

public DoublyLinkedList<E> getData() {
    return data;
}

public DoublyLinkedList<Edge<E>> getEdges() {
    return edges;
}

@Override
public String toString() {

    StringBuilder sb = new StringBuilder();
    int k = 0;
    for (int i = 0; i < sizeVertex(); i++) {
        if (vertices[i].getVertices().size() == 0) continue;
        if (vertices[i] == null) continue;
        sb.append(vertices[i] + " --> ");
        int j;

        for (j = k; j < edges.size(); j++) {
            if (String.valueOf(edges.get(j).getVer1()).equals(String.valueOf(vertices[i]))) {
                sb.append(edges.get(j).toString() + "; ");
            } else {
                break;
            }
        }
        k = j;
        sb.append(System.lineSeparator());
    }
    return sb.toString();
}
}

```

Package Graph.Algoritms

BreadthFirstSearch

```

package Graph.Algoritms;

import Graph.Graph;
import Graph.Vertex;
import MyStructures.DoublyLinkedList;
import MyStructures.Queue;

public class BreadthFirstSearch<E> {

    private final boolean[] marked;
    private Graph<E> graph ;
    private DoublyLinkedList<E> data;
    private final DoublyLinkedList<Vertex<E>> result;
    private E firstly;

    public BreadthFirstSearch(Graph<E> graph) {

        this.graph = graph;
        data = graph.getData();
        marked = new boolean[graph.getVertices().length];
        result = new DoublyLinkedList<>();

    }

    public void bfs(E item) {
        this.firstly = item;
        Queue<Vertex<E>> deque = new Queue<>();
        marked[data.indexOf(item)] = true;
        Vertex<E> currentV = new Vertex(item);
        deque.enqueue(currentV);
        while (!deque.isEmpty()) {
            E v = deque.dequeue().getObject();
            result.addLast(graph.getVertex(v));
            for (Vertex<E> w : graph.getVertex(v).getVertices()) {
                if (!marked[data.indexOf(w.getObject())]) {
                    deque.enqueue(w);
                    marked[data.indexOf(w.getObject())] = true;
                }
            }
        }
    }

    public void printRes() {
        System.out.print("\u001B[33m");
        System.out.print("BFS з вершини " + firstly + " : ");
        System.out.print("\u001B[0m");
        for (int i = 0; i < result.size(); i++) {
            System.out.print(result.get(i) + ", ");
        }
    }
}

```

```

    }
    System.out.println();
}
}

```

Package Graph.Algoritms DepthFirstSearch

```

package Graph.Algoritms;

import Graph.Graph;
import Graph.Vertex;
import MyStructures.DoublyLinkedList;

public class DepthFirstSearch<E> {

    private Graph<E> graph ;
    private final boolean[] marked;
    private final DoublyLinkedList<Vertex<E>> result;
    private DoublyLinkedList<E> data;
    private E firstly;
    int k = 0;

    public DepthFirstSearch(Graph<E> graph) {
        this.graph = graph;
        data = graph.getData();
        marked = new boolean[graph.sizeVertex()];
        result = new DoublyLinkedList<>();
    }

    public void dfs( E item) {

        if (k==0){
            firstly = item;
        }
        k++;

        marked[data.indexOf(item)] = true;
        result.addLast(graph.getVertex(item));
        for (Vertex<E> w : graph.getVertex(item).getVertices()) {
            if (!marked[data.indexOf(w.getObject())]) {
                dfs( w.getObject());
            }
        }
    }

    public void printRes() {
        System.out.print("\u001B[34m");
    }
}

```

```

        System.out.print("DFS з вершини " + firstly + " : ");
        System.out.print("\u001B[0m");
        for (int i = 0; i < result.size(); i++) {
            System.out.print(result.get(i) + ", ");
        }
        System.out.println();
    }
}

```

Package Graph.Algoritms Dijkstra

```

package Graph.Algoritms;

import Graph.Graph;
import Graph.Vertex;
import MyStructures.DoublyLinkedList;

import static java.lang.Integer.min;

public class Dijkstra<E> {
    private DoublyLinkedList<E> data;
    private Graph graph;

    private Vertex<E>[] vertices;
    private DoublyLinkedList<Integer> tarryingVertices;
    private int[] D;

    private E startElem;

    public Dijkstra(Graph graph) {
        this.graph = graph;
        this.data = graph.getData();
        this.vertices = graph.getVertices();
        this.D = new int[graph.getVertices().length];
    }

    public void dijkstraRes(E startElem, E elemFinish) {

        this.startElem = startElem;

        dijkstra(startElem);
        dijkstraTo(elemFinish);
        System.out.println(searchDist(startElem, elemFinish));
    }

    public void dijkstra(E startElem) {
        DoublyLinkedList<E> start = saveData();
    }
}

```



```

changeData(startElem);

D[0] = 0;
tarryingVertices = new DoublyLinkedList<>();

for (Vertex<E> v : vertices) {
    tarryingVertices.addLast(data.indexOf(v.getObject()));
}

tarryingVertices.remove(data.indexOf(data.get(0)));

for (int i = 1; i < vertices.length; i++) {
    D[i] = graph.getDistance(data.get(0), data.get(i));
}

while (!tarryingVertices.isEmpty()) {
    int minD = 1000000;
    int minV = 0;

    for (Integer v : tarryingVertices) {
        if (D[v] < minD) {
            minD = D[v];
            minV = v;
        }
    }

    tarryingVertices.remove(data.indexOf(data.get(tarryingVertices.indexOf(minV))));

    for (int v : tarryingVertices) {
        D[v] = min(D[v], D[minV] + graph.getDistance(data.get(v), data.get(minV)));
    }

}

data = start;
}

public void dijkstraTo(E elemFinish) {
    DoublyLinkedList<E> start = saveData();
    changeData(startElem);
    System.out.print("\u001B[33m");
    System.out.println("\nНайкоротший шлях від " + startElem + " до " + elemFinish);
    System.out.print("\u001B[0m");
    System.out.println(startElem + " --> " + elemFinish + " : " + D[data.indexOf(elemFinish)] + "
км");
    data = start;
}

public DoublyLinkedList<E> saveData() {

```

```

DoublyLinkedList<E> save = new DoublyLinkedList<>();
for (int i = 0; i < data.size(); i++) {
    save.addLast(data.get(i));
}
return save;
}

public void changeData(E elem) {
    DoublyLinkedList<E> res = new DoublyLinkedList<>();

    for (int i = data.indexOf(elem); i < vertices.length; i++) {
        res.addLast(data.get(i));
    }
    for (int j = 0; j < data.indexOf(elem); j++) {
        res.addLast(data.get(j));
    }
    for (int i = 0; i < data.size(); i++) {
        this.data.set(i, res.get(i));
    }
}

public DoublyLinkedList<E> searchDist(E startE, E finishE) {

    DoublyLinkedList<E> start = saveData();
    changeData(startElem);
    int result = D[data.indexOf(finishE)];
    data = start;

    DoublyLinkedList<E> res = new DoublyLinkedList<>();
    res.addLast(startE);
    Vertex<E> one = graph.getVertex(startE);

    for (Vertex<E> temp : one.getVertices()) {
        int w1 = graph.getEdge(one.getObject(), temp.getObject()).getWeight();

        if (w1 == result) {
            res.addLast(temp.getObject());
            return res;
        }
        for (Vertex<E> temp1 : temp.getVertices()) {
            int w2 = graph.getEdge(temp.getObject(), temp1.getObject()).getWeight();
            if (w1 + w2 == result && temp1.getObject().equals(finishE)) {
                res.addLast(temp.getObject());
                res.addLast(temp1.getObject());
                return res;
            }
        }
        for (Vertex<E> temp2 : temp1.getVertices()) {
            int w3 = graph.getEdge(temp1.getObject(), temp2.getObject()).getWeight();
            if (w1 + w2 + w3 == result && temp2.getObject().equals(finishE)) {
                res.addLast(temp.getObject());
            }
        }
    }
}

```

```

        res.addLast(temp1.getObject());
        res.addLast(temp2.getObject());
        return res;
    }

    for (Vertex<E> temp3 : temp2.getVertices()) {
        int w4 = graph.getEdge(temp2.getObject(), temp3.getObject()).getWeight();
        if (w1 + w2 + w3 + w4 == result && temp3.getObject().equals(finishE)) {
            res.addLast(temp.getObject());
            res.addLast(temp1.getObject());
            res.addLast(temp2.getObject());
            res.addLast(temp3.getObject());
            return res;
        }
    }
}

}

}
}
return res;
}
}

```

ReadInfoForFile

```

import Graph.Graph;
import MyStructures.BST;
import MyStructures.DoublyLinkedList;

import java.io.File;
import java.util.Scanner;

public class ReadInfoForFile {

    public BST<String, Integer> fillingBST() {
        BST<String, Integer> bst = new BST<>();
        try {
            Scanner sc = new Scanner(new
File("C:\\Users\\User\\IdeaProjects\\Cursach\\src\\resours\\info.csv"));
            sc.useDelimiter("\n");
            sc.next();

            while (sc.hasNext()) {
                String[] list = sc.next().split(",");

                bst.put(list[0], Integer.parseInt(list[1]));
            }
            sc.close();

        } catch (Exception e) {

```

```

        e.printStackTrace();
    }
    return bst;
}

public DoublyLinkedList<String> readCities() {

    DoublyLinkedList<String> cities = new DoublyLinkedList<>();

    try {
        Scanner sc = new Scanner(new
File("C:\\Users\\User\\IdeaProjects\\Cursach\\src\\resours\\info.csv"));
        sc.useDelimiter("\\n");
        sc.next();

        while (sc.hasNext()) {
            String[] list = sc.next().split(",");
            cities.addLast(String.valueOf(list[0]));

        }

        sc.close();

    } catch (Exception e) {
        e.printStackTrace();
    }

    return cities;
}

public void fillingGraph(Graph<String> graph) {

    try {
        Scanner sc = new Scanner(new
File("C:\\Users\\User\\IdeaProjects\\Cursach\\src\\resours\\info.csv"));
        sc.useDelimiter("\\n");
        sc.next();

        while (sc.hasNext()) {
            String[] list = sc.next().split(",");

            String firstCity = list[0];

            if (!list[2].equals("null")) {

                String[] connectedCities = list[2].split(";");

                String[] distance = list[3].split(";");

                for (int i = 0; i < connectedCities.length; i++) {

```

```

        graph.addEdge(String.valueOf(firstCity), String.valueOf(connectedCities[i]),
Integer.parseInt(distance[i]));
    }

    }
    }
    sc.close();
} catch (Exception e) {
    e.printStackTrace();
}
}
}

```

CarTask

```

import Graph.Graph;
import Graph.Edge;
import MyStructures.BST;
import MyStructures.DoublyLinkedList;

public class CarTask<E> {

    private final float massCar;
    private final float maxLoad;
    private final float fuelDiesel;
    private float fuelMass;
    private final float k;
    private DoublyLinkedList<Float> listSum;
    private Graph<E> graph;
    private BST<String, Integer> bst;

    public CarTask(float massCar, float maxLoad, float fuelDiesel, Graph<E> graph, BST<String,
Integer> bst) {
        this.massCar = massCar;
        this.maxLoad = maxLoad;
        this.fuelMass = massCar + maxLoad;
        this.fuelDiesel = fuelDiesel;
        this.graph = graph;
        this.bst = bst;
        this.listSum = new DoublyLinkedList<>();
        this.k = fuelDiesel / (massCar + maxLoad);
    }

    public float getDisel(float S, float mass) {

        if (fuelMass - mass < massCar) throw new NullPointerException("Вантаж закінчився");
        float res = (S / 100) * k * fuelMass;
        fuelMass -= mass;
        listSum.addLast(res);
    }
}

```

```

        return res;
    }

    public void printBaggage(){
        int counter = 0;
        System.out.print("\u001B[33m");
        System.out.print("Вантаж: ");
        System.out.print("\u001B[0m");
        for (String i :bst.keys()) {
            if (counter%3==0) System.out.println();
            System.out.print(bst.select(counter) + " --> ");
            System.out.print(bst.get(i) + " кг;\t");
            counter++;
        }
        System.out.println();

    }

    public float getDisel(DoublyLinkedList<E> list) {
        System.out.print("\u001B[34m");
        System.out.print("\nПрокладаємо маршрут: ");

        for (int i = 0; i < list.size() - 1; i++) {
            System.out.print(list.get(i) + " --> ");
        }
        System.out.println(list.get(list.size() - 1));
        System.out.print("\u001B[0m");

        DoublyLinkedList<Edge<E>> edges = new DoublyLinkedList<>();
        for (int i = 0; i < list.size() - 1; i++) {
            edges.addLast(graph.getEdge(list.get(i), list.get(i + 1)));
        }

        for (Edge<E> e : edges) {

            float S = (float) e.getWeight();
            float mass = (float) bst.get(e.getVer2().getObject().toString());
            System.out.printf("%s --> %s : %.2f км пройдено , %.2f кг вигружено, %.2f л  
витрачено\n", e.getVer1().getObject(), e.getVer2().getObject(), S, mass, getDisel(S, mass));
        }

        System.out.printf("Всього бензину: %.2f л\n", sumDiesel());
        return sumDiesel();

    }

    public void info() {
        System.out.print("\u001B[34m");

```

```

        System.out.println("\n\tHYUNDAI HD78");
        System.out.print("\u001B[0m");
        System.out.println("Маса автомобіля: " + massCar + " кг");
        System.out.println("Допустима маса вантажу: " + maxLoad + " кг");
        System.out.println("Обсяг дизелю на 100 км при повному завантажені: " + fuelDiesel + "
л\n");
    }

    public void resetDiesel() {
        listSum.clear();
    }

    public float sumDiesel() {
        float res = 0;
        for (int i = 0; i < listSum.size(); i++) {
            res += listSum.get(i);
        }
        return res;
    }
}

```

Main

```

import Graph.Algoritms.BreadthFirstSearch;
import Graph.Algoritms.DepthFirstSearch;
import Graph.Algoritms.Dijkstra;
import Graph.Graph;
import MyStructures.BST;
import MyStructures.DoublyLinkedList;

public class Main {
    public static void main(String[] args) {

        ReadInfoForFile read = new ReadInfoForFile();
        DoublyLinkedList<String> cities = read.readCities();
        BST<String,Integer> bst = read.fillingBST();

        Graph<String> graph = new Graph<>(cities);
        read.fillingGraph(graph);

        System.out.print("\u001B[33m");
        System.out.println("\t\tГраф міст заходу України");
        System.out.print("\u001B[0m");
        System.out.println(graph);
    }
}

```

```
BreadthFirstSearch<String> bfs = new BreadthFirstSearch<String>(graph);
bfs.bfs("Lutsk");
bfs.printRes();
```

```
DepthFirstSearch<String> dfs = new DepthFirstSearch<String>(graph);
dfs.dfs("Kyiv");
dfs.printRes();
```

```
Dijkstra<String> dijkstra = new Dijkstra<String>(graph);
dijkstra.dijkstraRes("Lutsk","Kyiv");
```

```
System.out.print("\u001B[34m");
System.out.println("\nВидалимо ребро Lutsk --> Rivne");
System.out.print("\u001B[0m");
graph.deleteEdge("Lutsk","Rivne");
```

```
dijkstra.dijkstraRes("Lutsk","Kyiv");
```

```
CarTask<String> car = new CarTask<String>(7500,5000,18,graph,bst);
car.info();
car.printBaggage();
car.getDisel(dijkstra.searchDist("Lutsk","Kyiv"));
```

```
}
```

```
}
```


Додаток 2. Файл info.csv

package Resours

```

1 Назва міста, Вага яку треба вивантажити в місто,З'єднанні міста,Довжина доріг до з'єднаних міст;
2 Lutsk,200,Lviv;Rivne,150;73;
3 Lviv,400,Uzhhorod;Ivano-Frankivsk;Ternopil;Rivne;Lutsk,248;133;145;211;150;
4 Rivne,500,Lutsk;Lviv;Ternopil;Khmeltsky;Zhytomyr,73;211;153;195;188;
5 Uzhhorod,700,Lviv;Ivano-Frankivsk,248;268;
6 Ivano-Frankivsk,900,Uzhhorod;Lviv;Ternopil;Chernivtsi,268;133;134;136;
7 Ternopil,800,Lviv;Rivne;Khmeltsky;Chernivtsi;Ivano-Frankivsk,145;153;111;172;134;
8 Zhytomyr,400,Rivne;Khmeltsky;Vinnytsia;Kyiv,188;217;129;151;
9 Khmeltsky,1000,Ternopil;Rivne;Zhytomyr;Vinnytsia;Chernivtsi,111;195;217;120;188;
10 Chernivtsi,600,Ivano-Frankivsk;Ternopil;Khmeltsky;Vinnytsia,136;172;188;272;
11 Kyiv,300,Zhytomyr;Vinnytsia,151;273;
12 Vinnytsia,300,Chernivtsi;Khmeltsky;Zhytomyr;Kyiv,272;120;129;273;
13 |

```