Міністерство освіти і науки України
Національний університет водного господарства та
Природокористування
Навчально-науковий інститут автоматики, кібернетики та
обчислювальної техніки
Кафедра комп'ютерних технологій та
економічної кібернетики

04-05-33

МЕТОДИЧНІ ВКАЗІВКИ

до виконання лабораторних робіт з навчальної дисципліни ПРОГРАМУВАННЯ (Частина 4. Нелінійні динамічні структури даних. Реалізація мовою програмування C#.) для здобувачів вищої освіти першого (бакалаврського) рівня за освітньо-професійною програмою «Інформаційні системи та технології» спеціальності 126 «Інформаційні системи та технології» та за освітньо-професійною програмою «Комп'ютерні технології» спеціальності 015 «Професійна освіта» денної та заочної форми навчання

Рекомендовано науково-методичною
радою з якості ННІ АКОТ
Протокол № 10 від 22.06.2020 р.

Рівне – 2020

Методичні вказівки до виконання лабораторних робіт з навчальної дисципліни ПРОГРАМУВАННЯ (Частина 4. Нелінійні динамічні структури даних. Реалізація мовою програмування C#) для здобувачів вищої освіти першого (бакалаврського) рівня за освітньо-професійною програмою «Інформаційні системи та технології» спеціальності 126 «Інформаційні системи та технології» денної форми навчання [Електронне видання] / Шевченко І. М. – Рівне : НУВГП, 2020. – 76 с.

Укладач: Шевченко І. М., старший викладач кафедри комп'ютерних технологій та економічної кібернетики.

Відповідальний за випуск: Грицюк П. М., д.е.н., професор, завідувач кафедри комп'ютерних технологій та економічної кібернетики.

Керівник групи забезпечення
спеціальності Гладка О. М.

Зміст

Лабораторна робота № 1. Поняття бінарного дерева. Обхід бінарного дерева. Створення, відображення дерева. Вставлення, видалення елементів у бінарному дереві.

Мета роботи: набути навичок створення та обробки бінарних дерев.

Послідовність виконання роботи:

1. Ознайомитись із теоретичними відомостями. (*Актуалізація опорних знань*).
2. Виконати програмування програм за поданими прикладами. Результат представити викладачу (*Застосування набутих знань*).
3. Виконати варіант самостійної роботи. (*Закріплення набутих знань*).
4. Оформити звіт на виконану роботу. (*Узагальнення та систематизація набутих знань*).
5. Захист звітів, відповіді на запитання.

1.1. Теоретичні відомості

Дерево – це структура даних, що являє собою сукупність елементів і відносин, що утворюють ієрархічну структуру цих елементів (рис. 1.1). Кожен елемент дерева називається вершиною (вузлом) дерева. Вершини дерева з'єднані спрямованими дугами, які називають гілками дерева. Початковий вузол дерева називають коренем дерева, йому відповідає нульовий рівень. Листями дерева називають вершини, в які входить одна гілка і не виходить жодної гілки.

Кожне дерево має такі властивості:

1) існує вузол, в який не входить ні одна дуга (корінь);
2) у кожну вершину, крім кореня, входить одна дуга.

Дерева особливо часто використовують на практиці при зображенні різних ієрархій. Наприклад, популярні

генеалогічні дерева.

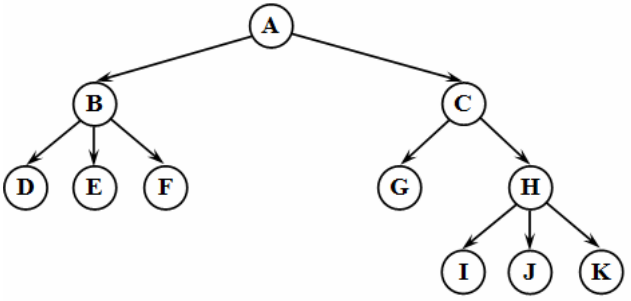


Рис. 1.1. Дерево

Всі вершини, до яких входять гілки, що виходять з однієї загальної вершини, називаються нащадками, а сама вершина − предком. Для кожного предка може бути виділено кілька нащадків. Рівень нащадка на одиницю перевищує рівень його предка. Корінь дерева не має предка, а листя дерева не мають нащадків.

Висота (глибина) дерева визначається кількістю рівнів, на яких розташовуються його вершини. Висота порожнього дерева дорівнює нулю, висота дерева з одного кореня − одиниці. На першому рівні дерева може бути тільки одна вершина − корінь дерева, на другому − нащадки кореня дерева, на третьому − нащадки нащадків кореня дерева і т.д.

Піддерево – частина деревовидної структури даних, яка може бути представлена у вигляді окремого дерева.

Ступенем вершини в дереві називається кількість дуг, які з неї виходить. Ступінь дерева дорівнює максимальному ступеню вершини, що входить у дерево. При цьому листями у дереві є вершини, що мають ступінь нуль. За величиною ступеня дерева розрізняють два типи дерев:

– бінарні − ступінь дерева не більше двох;
– сильнорозгалужені − ступінь дерева довільний.

Впорядковане дерево – це дерево, у якого гілки, що виходять з кожної вершини, впорядковані за певним

критерієм.

Дерева є рекурсивними структурами, тому що кожне піддерево також є деревом. Таким чином, дерево можна визначити як рекурсивну структуру, в якій кожен елемент є:

– або порожньою структурою;

– або елементом, з яким пов'язано кінцеве число піддерев.

Дії з рекурсивними структурами найзручніше описуються за допомогою рекурсивних алгоритмів.

Уявлення дерев, як списку, засноване на елементах, відповідних вершинам дерева. Кожен елемент має поле даних і два поля покажчиків: покажчик на початок списку нащадків вершини і покажчик на наступний елемент у списку нащадків поточного рівня. За такого способу подання дерева обов'язково слід зберігати покажчик на вершину, що є коренем дерева.

Для того, щоб виконати певну операцію над усіма вершинами дерева необхідно всі його вершини переглянути. Така дія називається обходом дерева.

Обхід дерева − це впорядкована послідовність вершин дерева, в якій кожна вершина зустрічається тільки один раз.

При обході всі вершини дерева повинні відвідуватися у певному порядку. Існує кілька способів обходу всіх вершин дерева. Виділимо три найбільш часто використовуваних способів обходу дерева (рис. 1.2):

– прямий;

– симетричний;

– зворотний

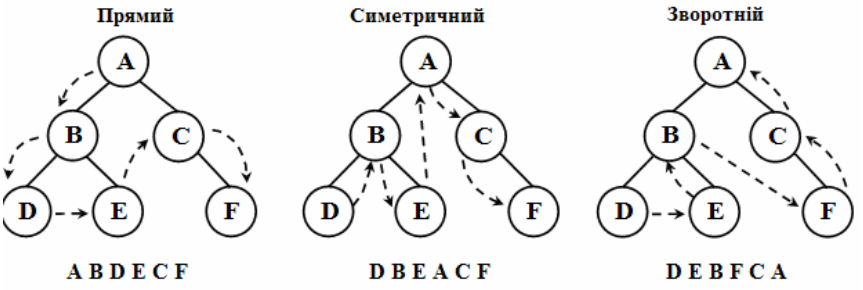


Рис. 1.2. Обхід дерев

Існує велике різноманіття деревовидних структур даних. Виділимо найпоширеніші з них: бінарні (двійкові) дерева, червоно-чорні дерева, B-дерева, АВЛ-дерева, матричні дерева, змішані дерева і т.д.

Бінарні дерева. Бінарні дерева є деревами зі ступенем не більше двох. Бінарне (двійкове) дерево − це динамічна структура даних, що являє собою дерево, в якому кожна вершина має не більше двох нащадків (рис. 1.3). Таким чином, бінарне дерево складається з елементів, кожен з яких містить інформаційне поле і не більше двох посилань на різні бінарні піддерева. На кожен елемент дерева є рівно одне посилання

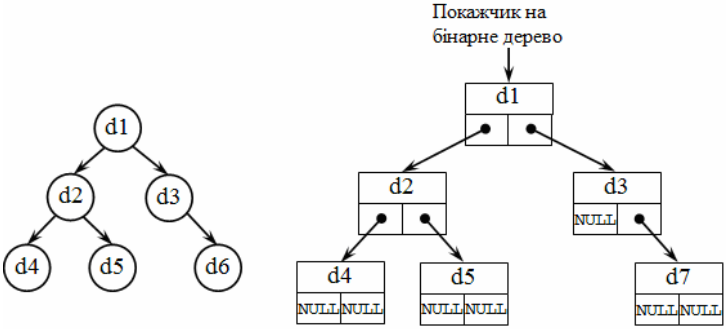


Рис. 1.3. − Бінарне дерево та його створення

Кожна вершина бінарного дерева є структурою, що складається з чотирьох видів полів. Вмістом цих полів будуть

відповідно:

- інформаційне поле (ключ вершини);
- службове поле (їх може бути декілька або жодного);
- покажчик на ліве піддерево;
- покажчик на праве піддерево.

За ступенем вершин бінарні дерева поділяються на (рис. 1.4):

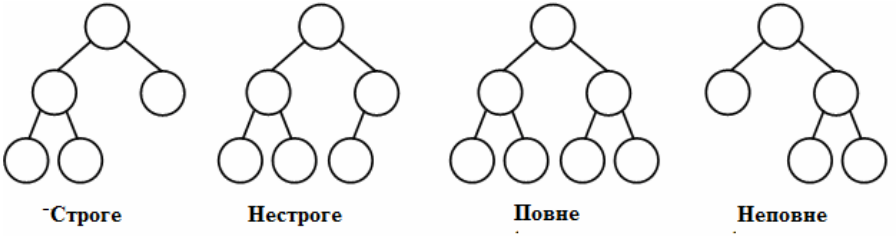


Рис. 1.4. Види бінарних дерев

- строге – вершини дерева мають ступінь нуль (у листях) або два (у вузлах);
- нестроге – вершини дерева мають ступінь нуль (у листях), один або два (у вузлах).

У загальному випадку у бінарного дерева на *k*-му рівні може бути до $2k–1$ вершин. Бінарне дерево називається *повним*, якщо воно містить тільки повністю заповнені рівні. В іншому випадку воно є *неповним.*

Дерево називається *збалансованим*, якщо довжини всіх шляхів від кореня до зовнішніх вершин рівні між собою. Дерево називається *майже збалансованим*, якщо довжини всіляких шляхів від кореня до зовнішніх вершин відрізняються не більше, ніж на одиницю.

Бінарне дерево може являти собою порожню множину. Бінарне дерево може виродитися у список (рис. 1.5).

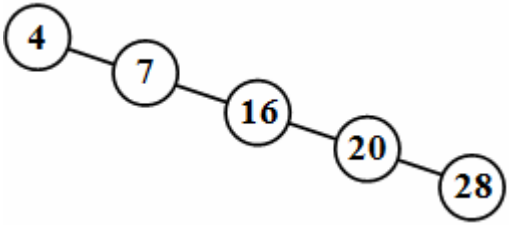


Рис. 1.5 Список як окремий випадок бінарного дерева

Структура дерева відбивається у вхідному потоці даних таким чином: кожному порожньому зв'язку, який додається, відповідає умовний символ, наприклад, '*' (зірочка). При цьому спочатку описуються ліві нащадки, потім праві. Для структури бінарного дерева, показанного на рис. 1.6, вхідний потік має вигляд: ABD * G *** CE ** FH ** J **.

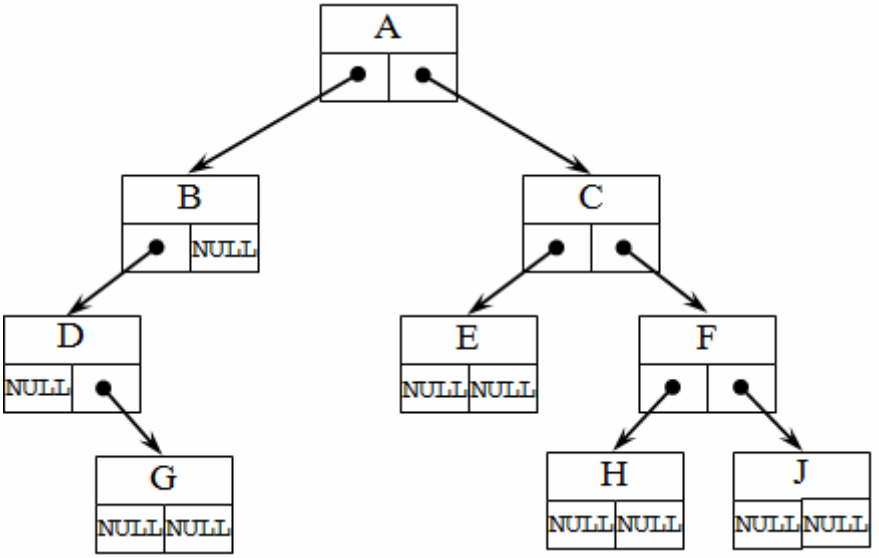


Рис. 1.6. Адресація у бінарному дереві

Бінарні дерева можуть застосовуватися для пошуку даних у спеціально побудованих деревах (бази даних), сортування даних, обчислень арифметичних виразів, кодування (метод

Хаффмана) і т.д.

Оголошення бінарного дерева виглядає таким чином:

```
public class ім’я типу
        {
            Інформаційне поле;
            [Службове поле;]
            Адреса лівого піддерева;
            Адреса правого піддерева;
        }
```

де інформаційне поле – це поле будь-якого раніше оголошеного або стандартного типу; адреса лівого (правого) піддерева - це покажчик на об'єкт того самого типу, яким і визначається структура, в нього записується адреса наступного елемента лівого (правого) піддерева.

Наприклад:

```
public class Tree
        {
            public int Data; // інформаційне поле
            public int count; // службове поле
            public Tree Left; // адреса лівого піддерева
            public Tree Right; // адреса правого піддерева
        }
```

Основними операціями, які здійснюються з бінарними деревами, є:

- створення бінарного дерева;
- друк бінарного дерева;
- обхід бінарного дерева;
- вставлення елемента в бінарне дерево;
- виключення елемента з бінарного дерева;
- перевірка порожнини бінарного дерева;
- видалення бінарного дерева.

Для опису алгоритмів цих основних операцій використовується наступне оголошення:

```
        public class Tree
        {
            public int Data;
            public Tree Left;
```

```
            public Tree Right;
        }
```

Дамо функції основних операцій при роботі з бінарним деревом.

```
    //Додавання нового елемента
    public static Tree AddNode(int inputDataNode, Tree root)
    {
        if (root == null)
        {
            root = new Tree();
            root.Data = inputDataNode;
            root.Left = null;
            root.Right = null;
        }
        else
        {
            if (inputDataNode < root.Data)
            {
                root.Left = AddNode(inputDataNode, root.Left);
            }
            else if (inputDataNode > root.Data)
            {
                root.Right = AddNode(inputDataNode, root.Right);
            }
        }
        return root;
    }
    // Друк дерева
    public static void PrintTree(int x, int y, Tree root, int
delta = 0)
    {
        if (root != null)
        {
            if (delta == 0) delta = x / 2;
            Console.SetCursorPosition(x, y);
            Console.Write(root.Data);
            PrintTree(x - delta, y + 3, root.Left, delta / 2);
            PrintTree(x + delta, y + 3, root.Right, delta / 2);
        }
    }
    // Симетричний обхід бінарного дерева
    public static void SimPrintTree(Tree root)
```

```
{
    if (root != null)
    {
        SimPrintTree(root.Left);
        Console.Write(root.Data+"  ");
        SimPrintTree(root.Right);
    }
}
// Прямий обхід бінарного дерева
public static void PrePrintTree(Tree root)
{
    if (root != null)
    {
        Console.Write(root.Data + "  ");
        PrePrintTree(root.Left);
        PrePrintTree(root.Right);
    }
}
// Зворотній обхід бінарного дерева
public static void PostPrintTree(Tree root)
{
    if (root != null)
    {
        PostPrintTree(root.Left);
        PostPrintTree(root.Right);
        Console.Write(root.Data + "  ");
    }
}
// Обрахунок суми елементів дерева
public static int SummaElements(Tree root)
{
    if (root == null)
        return 0;
    else
    {
        int count = 0;
        count += SummaElements(root.Left);
        count += SummaElements(root.Right);
        return count + root.Data;
    }
}
// Пошук елемента в бінарному дереві
    public static Tree FindElement(int findData, Tree root)
{
```

```
    if (root == null || findData == root.Data)
        return root;
    if (root.Data > findData)
        return FindElement(findData, root.Left);
    else
        return FindElement(findData, root.Right);
}

// Мінімальный элемент дерева
public static Tree Minimum(Tree root)
{
    Tree l=root;
    while (l.Left != null)
    {
        l=l.Left;
    }
    return l;
}

// Максимальний елемент дерева
public static Tree Maximum(Tree root)
{
    Tree r = root;
    while (r.Right != null)
        r = r.Right;
    return r;
}
// Видалення вузла бінарного дерева
public static Tree DeleteNode(int deleteData, Tree root)
{
    if (root == null)
        return root;
    if (deleteData < root.Data)
    {
        root.Left = DeleteNode(deleteData, root.Left);
    }
    else if (deleteData > root.Data)
    {
        root.Right = DeleteNode(deleteData, root.Right);
    }
    else if (root.Left != null && root.Right != null)
    {
        root.Data = Minimum(root.Right).Data;
        root.Right = DeleteNode(root.Data, root.Right);
```

```
        }
        else if (root.Left != null)
        {
            root=root.Left;
        }
        else if (root.Right != null)
        {
            root= root.Right;
        }
        else
            root = null;
        return root;
    }
```

1.2. Індивідуальні завдання

Потрібно реалізувати кожне завдання у відповідності з наведеними етапами:

1. вивчити словесну постановку задачі, виділяючи при цьому всі види даних;
2. сформулювати математичну постановку задачі;
3. обрати метод розв’язання задачі, якщо це необхідно;
4. розробити графічну схему алгоритму;
5. записати розроблений алгоритм мовою програмування;
6. розробити контрольний тест до програми;
7. налагодити програму;
8. представити звіт до роботи.

Варіант кожного завдання обирається за номером студента в журналі.

Варіант	Використовуваний спосіб обходу дерева для виконання завдання	Завдання
1.	прямий	1. Знайти суму всіх елементів дерева 2. Вивести на екран всі листи дерева

2.	зворотній	1. Обчислити середнє арифметичне елементів дерева 2. Вивести на екран ті листи дерева, які мають парні значення
3.	симетричний	1. Знайти кількість парних елементів дерева 2. Вивести на екран значення вузлів, у яких тільки один нащадок (лівий або правий)
4.	прямий	1. Обчислити кількість елементів дерева, кратних 10 2. Видалити ті листи в дереві, у яких немає братів
5.	зворотній	1.Збільшити всі елементи дерева в два рази 2. Визначити, чи є дерево майже повним
6.	симетричний	1. Всі від'ємні елементи дерева замінити на нулі 2. Вивести на екран ті вузли, сума значень синів яких парна
7.	прямий	1. Реверсувати кожен елемент дерева, тобто якщо є елемент 345, то замінюємо його на 543 2. Визначити в якому піддереві (лівому або правому) кількість парних елементів більша. Вивести ці елементи на екран для кожного піддерева окремо
8.	зворотній	1. Визначити кількість елементів дерева, що закінчуються на цифру 9 2. Вивести перелік тих вузлів, у яких кількість парних елементів в правому і лівому піддереві рівні.

9.	симетричний	1.Ті елементи дерева, які складаються з однакових цифр, замінити значенням 0 2.Вивесті на екран ті вузли, у яких різниця сум елементів в правому і лівому поддереві більша 20. Вивести ці суми для кожного вузла

1.3. Зміст|вміст| звіту:

1. Прізвище та ім'я студента.
2. Номер і назва лабораторної роботи.
3. Мета роботи.
4. Номер індивідуального завдання. Текст завдання (постановка завдання).
5. Лістинг програми.
6. Скриншот робочої програми.
7. Висновок про засвоєнні знання та вміння.

1.4. Питання для самоперевірки

1. З чим пов'язана популярність використання дерев у програмуванні?
2. Чи можна список віднести до дерев? Відповідь обґрунтуйте.
3. Які дані містять адресні поля елемента бінарного дерева?
4. Чи може бінарне дерево бути строгим і неповним? Відповідь обґрунтуйте.
5. Чи може бінарне дерево бути нестрогим і повним? Відповідь обґрунтуйте.
6. Яким може бути майже збалансоване бінарне дерево: повним, неповним, строгим, нестрогим? Відповідь обґрунтуйте.
7. Куди може бути доданий елемент у бінарне дерево у

залежності від його виду (повне, неповне, строгим, нестрогим)? Вид дерева при цьому повинен зберегтися.

8. Куди може бути доданий елемент у збалансоване бінарне дерево? Вид дерева при цьому повинен зберегтися.
9. Чим відрізняються, з точки зору реалізації алгоритму прямий, симетричний і зворотний обходи бінарного дерева?
10. На підставі чого в червоно-чорному дереві найдовша гілка від кореня до листя не більше ніж удвічі довше будь-який іншої гілки від кореня до листя?
11. Як можна охарактеризувати червоно-чорне дерево: повне, неповне, строгим, нестрогим?
12. Яким чином при видаленні елемента з червоно-чорного дерева перефарбовуються вузли?

Лабораторна робота № 2. Графи. Подання графів у програмуванні. Алгоритми пошуку оптимальних шляхів у графах.

Мета роботи: набути ||навичок| створення .

Послідовність виконання роботи:

1. Ознайомитись із теоретичними відомостями. (*Актуалізація опорних знань*).
2. Виконати програмування програм за поданими прикладами. Результат представити викладачу (*Застосування набутих знань*).
3. Виконати варіант самостійної роботи. (*Закріплення набутих знань*).
4. Оформити звіт на виконану роботу. (*Узагальнення та систематизація набутих знань*).
5. Захист звітів, відповіді на запитання.

2.1. Теоретичні відомості

Граф – це сукупність не порожньої множини V вершин і множини E невпорядкованих або впорядкованих пар вершин: G=(V,E), $V \neq \varnothing$, $E \subset V \times V$. Невпорядкована пара вершин називається ребром, впорядкована пара вершин – дугою. При цьому першу вершину з пари прийнято називати початком дуги, а другу – її кінцем.

Граф, що не містить дуг, називається неорієнтованим, а граф, що містить дуги – орієнтованим або орграфом.

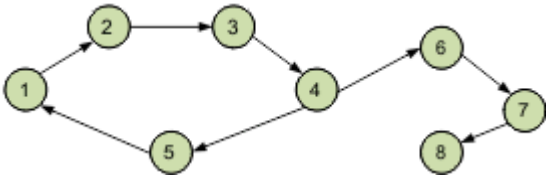


Рис. 2.1. Орієнтований граф.

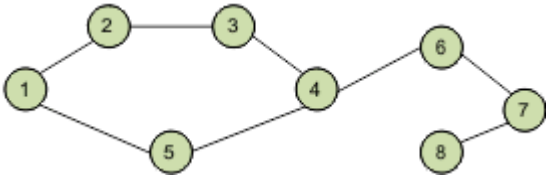


Рис. 2.2. Неорієнтований граф.

В орієнтованому графі ребра є спрямованими, тобто існує тільки однин доступний напрямок між двома зв'язаними вершинами.

У неорієнтованому графі по кожному з ребер можна здійснити перехід в обох напрямках.

Окремий випадок двох цих видів – змішаний граф. Він характерний наявністю як орієнтованих, так і неорієнтовані ребер.

Ребро (дуга) і будь-яка його вершина називаються інцидентними. З'єднані ребром вершини називаються суміжними. Якщо вершина *v* є початком певної дуги, а вершина *w* – її кінцем, то вершину *w* вважають суміжною з вершиною *v*, але не навпаки.

Ступінь входу вершини – кількість вхідних в неї ребер, ступінь виходу – кількість вихідних ребер. Граф, що містить ребра між усіма парами вершин, є повним. Зустрічаються такі графи, ребрах яких поставлено у відповідність конкретне числове значення, вони називаються виваженими графами, а це значення – вагою ребра.

Якщо у ребра обидва кінці збігаються, тобто вони виходять з вершини і входить в неї, то таке ребро називається петлею.

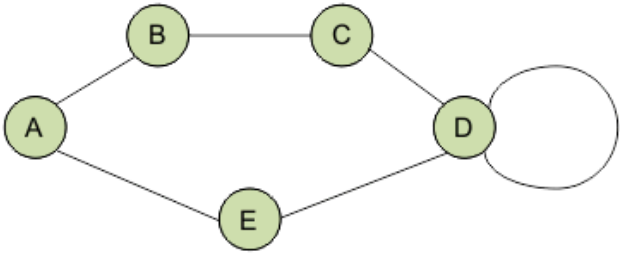
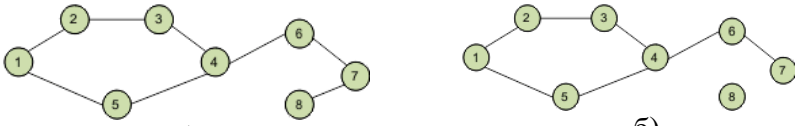


Рис. 2.3. Неорієнтований граф з петлею.

Графи також поділяються на зв’язні та незв’язні:



а) б)

Рис. 2.4. Зв’язний (а) та незв’язний (б) графи.

У зв'язному графі між будь-якою парою вершин існує як мінімум один шлях. У незв'язному графі існує хоча б одна вершина, не пов'язана з іншими.

Для використання графа у програмування потрібно обрати спосіб їх зображення в оперативній пам’яті. Граф може бути представлений (збережений) декількома способами:

1. матриця суміжності;
2. матриця інцидентності;
3. список суміжності (інцидентності);
4. список ребер.

Використання двох перших методів передбачає зберігання графа у вигляді двовимірного масиву (матриці). Розмір масиву залежить від кількості вершин і / або ребер в конкретному графі.

Нехай n – кількість вершин графу, m – кількість його ребер. Вважаємо, що всі вершини графу пронумеровані натуральними числами від 1 до n, а всі ребра – натуральними числами від 1 до m. Матриця суміжності Adj є квадратною матрицею розмірності $n \times n$, в який

$$Adj[i,j] = \begin{cases} 1, \textit{вершина } j \textit{ суміжна з вершиною } i \\ 0, \textit{вершина } j \textit{ не суміжна з вершиною } i \end{cases}$$

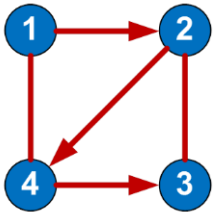
Матриця інциденції Inc неорієнтованого графу є матрицею розмірності $n \times m$, в якій

$$Inc[i,j] = \begin{cases} 1, \textit{вершина } v_i \textit{ інцидентна ребру } e_j \\ 0, \textit{вершина } v_i \textit{ неінцидентна ребру } e_j \end{cases}$$

Матриця інциденції Inc орграфу є матрицею розмірності $n \times m$, в якій

$$Inc[i,j] = \begin{cases} 1, \textit{вершина } v_i \textit{ інцидентна дузі } e_j \textit{ і є її кінцем} \\ 0, \textit{вершина } v_i \textit{ неінцидентна ребру } e_j \\ -1, \textit{вершина } v_i \textit{ інцидентна дузі } e_j \textit{ і є її початком} \end{cases}$$

Приклад 2.1. Побудувати матрицю суміжності для графа:

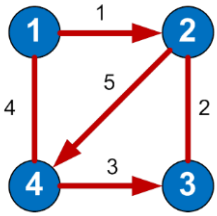


Якщо з однієї вершини в іншу прохід вільний (існує ребро), в осередок заноситься 1, інакше - 0. Всі елементи на головній діагоналі рівні 0, якщо граф не має петель.

Матриця суміжності буде мати вигляд:

	1	2	3	4
1	0	1	0	1
2	0	0	1	1
3	0	1	0	0
4	1	0	1	0

Приклад 2.2. Побудувати матрицю інциденції для графа:



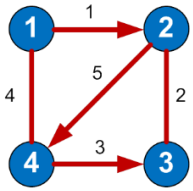
Матриця інцидентності (інциденцій) графа – це матриця, кількість рядків в якій відповідає числу вершин, а кількість стовпців – числу ребер. У ній вказуються зв'язок між інцидентними елементами графа (ребро (дуга) і вершина). У неорієнтованому графі якщо вершина інцидентна ребру то відповідний елемент дорівнює 1, в іншому випадку елемент дорівнює 0.

В орієнтованому графі якщо ребро виходить з вершини, то відповідний елемент дорівнює 1, якщо ребро входить в вершину, то відповідний елемент дорівнює -1, якщо ребро відсутня, то елемент дорівнює 0.

	1	2	3	4	5
1	1	0	0	1	0
2	-1	1	0	0	1
3	0	1	-1	0	0
4	0	0	1	1	-1

Матриця інцидентності для свого представлення вимагає нумерації ребер, що не завжди зручно.

Приклад 2.3. Побудувати список суміжності (інциденції) для графа:



Якщо кількість ребер графа в порівнянні з кількістю вершин невелика, то значення більшості елементів матриці суміжності дорівнюватимуть 0. При цьому використання даного методу недоцільно. Для подібних графів є більш оптимальні способи їх подання.

По відношенню до пам'яті списки суміжності менш вимогливі, ніж матриці суміжності. Такий список можна представити у вигляді таблиці, в якій 2 стовпці, а рядків – не більш, ніж вершин в графі.

В кожному рядку в першому стовпці вказана вершина виходу, а в другому стовпці – список вершин, в які входять ребра з поточної вершини.

1	2, 4
2	3, 4
3	2
4	1, 3

Переваги списку суміжності:

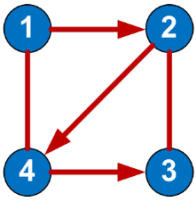
– Раціональне використання пам'яті.
– Дозволяє швидко перебирати сусідів вершини.
– Дозволяє перевіряти наявність ребра і видаляти його.

Недоліки списку суміжності:

– При роботі з насиченими графами (з великою кількістю ребер) швидкості може не вистачати.
– Немає швидкого способу перевірити, чи існує ребро між двома вершинами.

– Кількість вершин графа має бути відомо заздалегідь.
– Для зважених графів доводиться зберігати список, елементи якого повинні містити два значущих поля, що ускладнює код:
    - номер вершини, з якої з'єднується поточна;
    - вага ребра.

Приклад 2.4. Побудувати список ребер для графа:



В списку ребер в кожному рядку записуються дві суміжні вершини і вага з'єднує їх ребра (для зваженого графа). Кількість рядків у списку ребер завжди має дорівнювати величині, що виходить в результаті складання орієнтованих ребер з подвоєною кількістю неорієнтованих ребер.

	Начало	Конец	Вес
1	1	2	
2	1	4	
3	2	3	
4	2	4	
5	3	2	
6	4	1	
7	4	3	

Який спосіб представлення графа краще? Відповідь залежить від співвідношення між числом вершин і числом ребер. Число ребер може бути досить малим (такого ж порядку, як і кількість вершин) або досить великим (якщо граф є повним). Графи з великим числом ребер називають щільними, з малим – розрідженими. Щільні графи зручніше зберігати в вигляді матриці суміжності, розріджені – у вигляді списку суміжності.

Алгоритми обходу графа. Основними алгоритмами обходу графа є

- пошук в ширину;
- пошук в глибину.

При **пошуку в ширину**, після відвідин першої вершини, відвідуються всі сусідні з нею вершини. Потім відвідуються всі вершини, що знаходяться на відстані двох ребер від початкової. При кожному новому кроці відвідуються вершини, відстань від яких до початкової на одиницю більше попереднього. Щоб запобігти повторному відвідування вершин, необхідно вести список відвіданих вершин. Для зберігання тимчасових даних, необхідних для роботи алгоритму, використовується черга – впорядкована послідовність елементів, в якій нові елементи додаються в кінець, а старі видаляються з початку.

Таким чином, основна ідея пошуку в ширину полягає в тому, що спочатку досліджуються всі вершини, суміжні з початковою вершиною (вершина з якої починається обхід). Ці вершини знаходяться на відстані 1 від початкової. Потім досліджуються всі вершини на відстані 2 від початкової, потім всі на відстані 3 і т.д. Звернемо увагу, що при цьому для кожної вершини відразу знаходяться довжина найкоротшого маршруту від початкової вершини.

Алгоритм пошуку в ширину

Крок 1. Всім вершинам графа присвоюється значення «не відвідана». Вибирається перша вершина і позначається як «відвідана» (і заноситься в чергу).

Крок 2. Відвідується перша вершина з черги (якщо вона не позначена як відвідана). Всі її сусідні вершини заносяться в чергу. Після цього вона видаляється з черги.

Крок 3. Повторюється крок 2 до тих пір, поки черга не порожня (рис. 44.6).

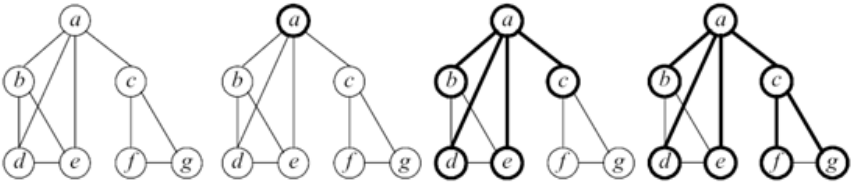


Рис. 2.5. Обхід графа в ширину

Застосування алгоритму пошуку в ширину

- Пошук найкоротшого шляху в незваженому графі (орієнтованому або неорієнтованому).
- Пошук компонент зв'язності.
- Знаходження розв’язку будь-якої задачі (гри) з найменшим числом ходів.
- Знаходження всіх ребер, що лежать на якомусь найкоротшому шляху між заданою парою вершин.
- Знаходження всіх вершини, що лежать на якомусь найкоротшому шляху між заданою парою вершин.

Алгоритм пошуку в ширину працює як на орієнтованих, так і на неорієнтованих графах. Для реалізації алгоритму зручно використовувати чергу.

При **пошуку в глибину** відвідується перша вершина, потім необхідно йти вздовж ребер графа, до попадання в глухий кут. Вершина графа є тупиком, якщо все суміжні з нею вершини вже відвідані. Після потрапляння в глухий кут потрібно повертатися назад уздовж пройденого шляху, поки не буде виявлена вершина, у якій є ще не відвідана вершина, а потім необхідно рухатися в цьому новому напрямку. Процес виявляється завершеним при поверненні в початкову вершину, причому всі суміжні з нею вершини вже повинні бути відвідані.

Таким чином, основна ідея пошуку в глибину – коли можливі шляхи по ребрах, що виходять з вершин, розгалужуються, потрібно спочатку повністю дослідити одну гілку і тільки потім переходити до інших гілок (якщо вони залишаться нерозглянутими).

Алгоритм пошуку в глибину

Крок 1. Всім вершинам графа присвоюється значення «не відвідана». Вибирається перша вершина і позначається як відвідана.

Крок 2. Для останньої поміченої як відвідана вершини вибирається суміжна вершина, яка є першою поміченої що не відвідана, і їй присвоюється значення відвідана. Якщо таких вершин немає, то береться попередня позначена вершина.

Крок 3. Повторити крок 2 до тих пір, поки всі вершини не будуть позначені як відвідані (рис. 2.6).

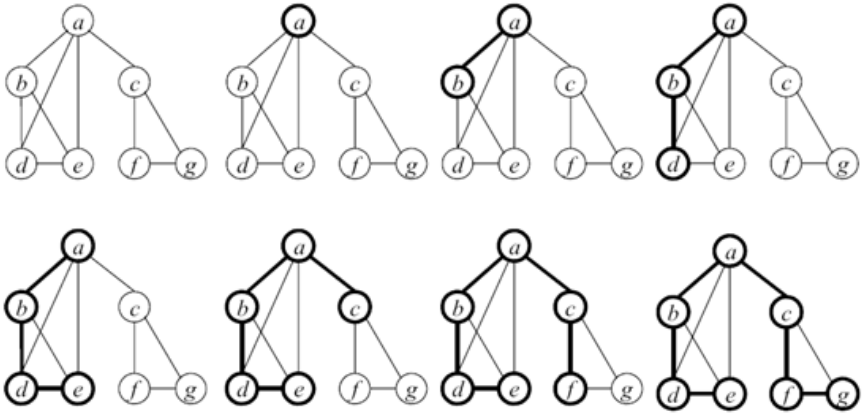


Рис. 2.5. Обхід графа в глибину

Застосування алгоритму пошуку в глибину:

- Пошук будь-якого шляху в графі.
- Пошук лексикографічно першого шляху в графі.
- Перевірка, чи є одна вершина дерева предком іншої.
- Пошук найменшого спільного предка.
- Топологічне сортування.
- Пошук компонент зв'язності.

Алгоритм пошуку в глибину працює як на орієнтованих, так і на неорієнтованих графах. Застосованість алгоритму залежить від конкретного завдання.Для реалізації алгоритму зручно використовувати стек або рекурсію.

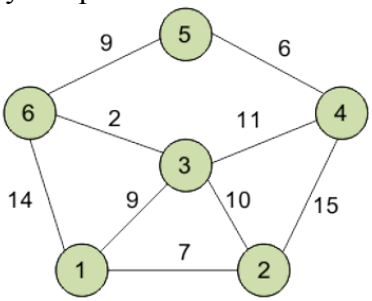
Приклад використання графів – Алгоритм Дейкстри знаходження найкоротшого шляху у зважених графах.

Розглянемо приклад знаходження найкоротшого шляху. Дана мережа автомобільних доріг, що з'єднують населені пункти області. Деякі дороги односторонні. Знайти найкоротші шляхи від центру міста до кожного міста області.

Для вирішення цього завдання можна використовувати алгоритм Дейкстри - алгоритм на графах, винайдений нідерландським вченим Е. Дейкстрой в 1959 році. Знаходить найкоротшу відстань від однієї з вершин графа до всіх інших. Працює тільки для графів без ребер з негативною вагою.

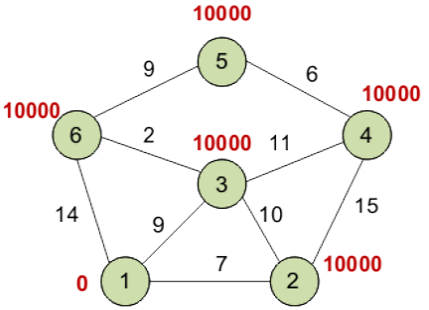
Нехай потрібно знайти найкоротший відстані від 1-ї вершини до всіх інших.

Кружочками позначені вершини, лініями – шляхи між ними (ребра графа). У кружочках позначені номери вершин, над ребрами позначений їх вага – довжина шляху. Поряд з кожною вершиною червоним позначена мітка – довжина найкоротшого шляху в цю вершину з вершини 1.



Ініціалізація

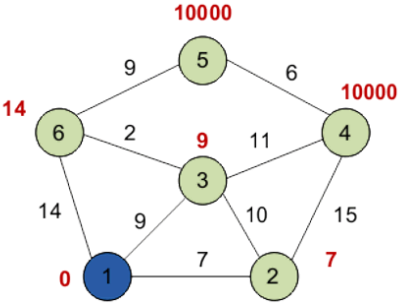
Мітка самої вершини 1 покладається рівною 0, мітки інших вершин – недосяжно велике число (в ідеалі – нескінченність). Це відображає те, що відстані від вершини 1 до інших вершин поки невідомі. Всі вершини графа позначаються як невідвідані.



Перший крок

Мінімальну мітку має вершина 1. Її сусідами є вершини 2, 3 і 6. Обходимо сусідів вершини по черзі.

Перший сусід вершини 1 – вершина 2, тому що довжина шляху до неї мінімальна. Довжина шляху в неї через вершину 1 дорівнює сумі найкоротшої відстані до вершини 1, значенням її мітки, і довжини ребра, що йде з 1-й в 2-ю, тобто 0 + 7 = 7. Це менше поточної мітки вершини 2 (10000) , тому нова мітка 2-й вершини дорівнює 7.

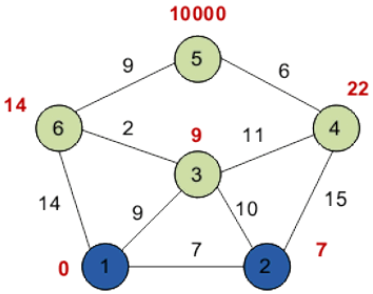


Аналогічно знаходимо довжини шляху для всіх інших сусідів (вершини 3 і 6). Всі сусіди вершини 1 перевірені. Поточне мінімальна відстань до вершини 1 вважається остаточним і перегляду не підлягає. Вершина 1 позначається як відвідана.

Другий крок

Крок 1 алгоритму повторюється. Знову знаходимо «найближчу» з невідвіданих вершин. Це вершина 2 з міткою 7. Знову намагаємося зменшити мітки сусідів обраної вершини, намагаючись пройти в них через 2-ю вершину. Сусідами

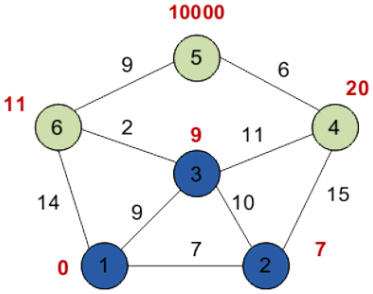
вершини 2 є вершини 1, 3 і 4. Вершина 1 вже переглянуло. Наступний сусід вершини 2 – вершина 3, так як має мінімальну позначку з вершин, позначених що не відвідані. Якщо йти в неї через 2, то довжина такого шляху буде дорівнює 17 (7 + 10 = 17). Але поточна мітка третьої вершини дорівнює 9, а 9 <17, тому мітка не змінюється.



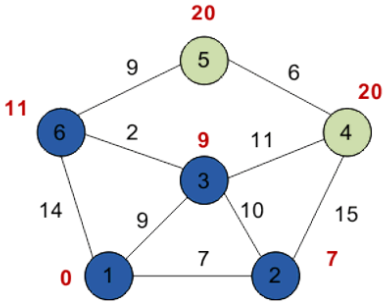
Ще один сусід вершини 2 – вершина 4. Якщо йти в неї через 2-ю, то довжина такого шляху буде дорівнює 22 (7 + 15 = 22). Оскільки 22 <10000, становлюємо мітку вершини 4 рівній 22. Всі сусіди вершини 2 переглянуті, помічаємо її як відвідану.

Третій крок

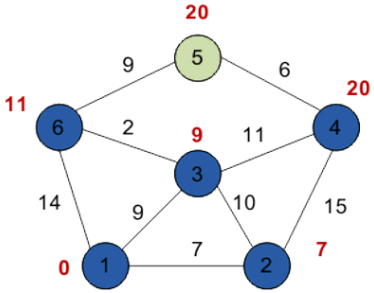
Повторюємо крок алгоритму, вибравши вершину 3. Після її «обробки» отримаємо наступні результати.



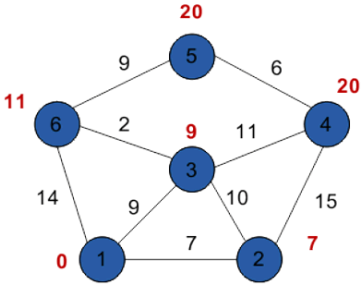
Четвертий крок



П’ятий крок



Шостий крок



Таким чином, найкоротшим шляхом з вершини 1 у вершину 5 буде шлях через вершини 1 - 3 - 6 - 5, оскільки таким шляхом ми набираємо мінімальну вагу, рівну 20.

Займемося виведенням найкоротшого шляху. Ми знаємо довжину шляху для кожної вершини, і тепер будемо розглядати вершини з кінця. Розглядаємо кінцеву вершину (в даному випадку – вершина 5), і для всіх вершин, з якою вона пов'язана, знаходимо довжину шляху, віднімаючи вагу відповідного ребра з довжини шляху кінцевої вершини.

Так, вершина 5 має довжину шляху 20. Вона пов'язана з вершинами 6 і 4.

Для вершини 6 отримаємо вагу 20 - 9 = 11 (збіглася).

Для вершини 4 отримаємо вагу 20 - 6 = 14 (не співпала).

Якщо в результаті ми отримаємо значення, яке збігається з довжиною шляху розглянутої вершини (в даному випадку – вершина 6), то саме з неї був здійснений перехід в кінцеву вершину. Відзначаємо цю вершину на шуканому шляху. Далі визначаємо ребро, через яке ми потрапили в вершину 6. І так поки не дійдемо до початку. Якщо в результаті такого обходу у нас на якомусь етапі співпадуть значення для декількох вершин, то можна взяти будь-яку з них – кілька шляхів матимуть однакову довжину.

Реалізація алгоритму Дейкстри

Для зберігання вагів графа використовується квадратна матриця. У заголовках рядків і стовпців знаходяться вершини графа. Ваги дуг графа розміщуються у внутрішніх осередках таблиці. Граф не містить петель, тому на головній діагоналі матриці містяться нульові значення.

	1	2	3	4	5	6
1	0	7	9	0	0	14
2	7	0	10	15	0	0
3	9	10	0	11	0	2
4	0	15	11	0	6	0
5	0	0	0	6	0	9
6	14	0	2	0	9	0

Реалізація алгоритму мовою програмування:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp19
{
```

```
    class Program
    {
        public const int SIZE = 6;
        static void Main(string[] args)
        {
            int[,] a = new int[SIZE, SIZE]; // матриця зв'язків
            int[] d = new int[SIZE]; // мінімальна відстань
            int[] v = new int[SIZE]; // відвідувані вершини
            int temp, minindex, min;
            int begin_index = 0;
            // Ініціалізація матриці зв'язків
            for (int i = 0; i < SIZE; i++)
            {
                a[i, i] = 0;
                for (int j = i + 1; j < SIZE; j++)
                {
                    Console.Write("Введіть відстань {0} - {1}:
", i + 1, j + 1);
                    a[i, j] = a[j, i] =
Convert.ToInt32(Console.ReadLine());
                }
            }
            // Вивід матриці зв'язків
            Console.WriteLine("\nВиведення матриці зв'язків ");
            for (int i = 0; i < SIZE; i++)
            {
                for (int j = 0; j < SIZE; j++)
                    Console.Write("{0} ", a[i, j]);
                Console.WriteLine();
            }
            //Ініціалізація вершин і відстаней
            for (int i = 0; i < SIZE; i++)
            {
                d[i] = 10000;
                v[i] = 1;
            }
            d[begin_index] = 0;
            // Крок алгоритма
            do
            {
                minindex = 10000;
                min = 10000;
                for (int i = 0; i < SIZE; i++)
                { // Якщо вершину ще не обішли і вага менше min
```

```
                    if ((v[i] == 1) && (d[i] < min))
                    { // Переприсвоюємо значення
                        min = d[i];
                        minindex = i;
                    }
                }
                // Додаємо знайдену мінімальну вагу до поточної
ваги вершини
                // і порівнюємо з поточною мінімальною вагою
вершини
                if (minindex != 10000)
                {
                    for (int i = 0; i < SIZE; i++)
                    {
                        if (a[minindex, i] > 0)
                        {
                            temp = min + a[minindex, i];
                            if (temp < d[i])
                            {
                                d[i] = temp;
                            }
                        }
                    }
                    v[minindex] = 0;
                }
            } while (minindex < 10000);
            // Виведення найкоротшої відстані до вершин
            Console.WriteLine("\nНайкоротші відстані до вершини:
");
            for (int i = 0; i < SIZE; i++)
                Console.Write("{0} ", d[i]);
            // Відновлення шляху
            int[] ver = new int[SIZE]; // масив відвіданих
вершин
            int end = 4; // індекс кінцевої вершини = 5 - 1
            ver[0] = end + 1; // початковий елемент - кінцева
вершина
            int k = 1; // індекс попередньої вершин
            int weight = d[end]; // вага кінцевої вершини
            while (end != begin_index) // поки не дійшли до
початкової вершини
            {
                for (int i = 0; i < SIZE; i++) // переглядаємо
всі вершини
```

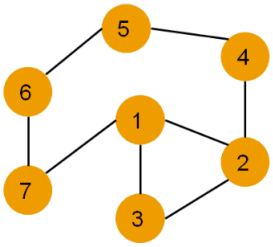
```
                    if (a[end, i] != 0)   // якщо зв'язок є
                    {
                        int temp1 = weight - a[end, i]; //
визначаємо вагу шляху з попередньої вершини
                        if (temp1 == d[i]) // якщо вага співпала
з розрахованою,
                        {                 //то перехід був
зроблений з цієї вершини
                            weight = temp1; // зберігаємо нову
вагу
                            end = i;       // зберігаємо
попереднюю вершину
                            ver[k] = i + 1; // та записуємо її в
масив
                            k++;
                        }
                    }
            }
            // Виведення шляху () початкова вершина опинилася в
кінці масиву з k елементів)
            Console.WriteLine("\n\nВиведення найкоротшого
шляху");
            for (int i = k - 1; i >= 0; i--)
                Console.Write("{0} ", ver[i]);
            Console.ReadLine();
        }
    }
}
```

Результат роботи програми:

```
C:\Users\Администратор\source\repos\Console
Введіть відстань 1 - 5: 0
Введіть відстань 1 - 6: 14
Введіть відстань 2 - 3: 10
Введіть відстань 2 - 4: 15
Введіть відстань 2 - 5: 0
Введіть відстань 2 - 6: 0
Введіть відстань 3 - 4: 11
Введіть відстань 3 - 5: 0
Введіть відстань 3 - 6: 2
Введіть відстань 4 - 5: 6
Введіть відстань 4 - 6: 0
Введіть відстань 5 - 6: 9

Виведення матриці зв'язків
0 7 9 0 0 14
7 0 10 15 0 0
9 10 0 11 0 2
0 15 11 0 6 0
0 0 0 6 0 9
14 0 2 0 9 0

Найкоротші відстані до вершини:
0 7 9 20 20 11

Виведення найкоротшого шляху
1 3 6 5
```

2.2. Приклади виконання завдань

Приклад 2.2.1. Написати програму для обходу графа в ширину.



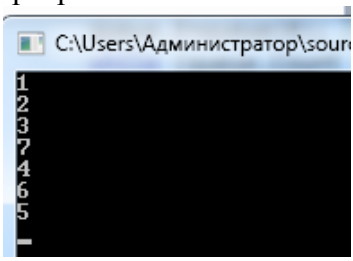
Граф представлений матрицею суміжності:

0	1	1	0	0	0	1
1	0	1	1	0	0	0
1	1	0	0	0	0	0
0	1	0	0	1	0	0
0	0	0	1	0	1	0
0	0	0	0	1	0	1
1	0	0	0	0	1	0

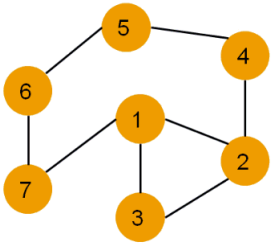
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp15
{
    class Program
    {
        static void Main(string[] args)
        {
            Queue<int> queue = new Queue<int>();
            // матрица суміжності
           int[,] mas = new int[7, 7]{{ 0, 1, 1, 0, 0, 0, 1 },
                                      { 1, 0, 1, 1, 0, 0, 0 },
                                      { 1, 1, 0, 0, 0, 0, 0 },
                                      { 0, 1, 0, 0, 1, 0, 0 },
                                      { 0, 0, 0, 1, 0, 1, 0 },
                                      { 0, 0, 0, 0, 1, 0, 1 },
                                      { 1, 0, 0, 0, 0, 1, 0 } };
            int[] nodes = new int[7]; // вершини графа
            for (int i = 0; i < 7; i++)
                nodes[i] = 0; // на початку всі вершини равны 0
(не відвідані)
            queue.Enqueue(0); // розміщуємо в чергу першу
вершину
            while (queue.Count != 0)
            { // поки черга не порожня
                int node = queue.Dequeue(); // витягаємо вершину
                nodes[node] = 2; // помічаємо її як відвідану
                for (int j = 0; j < 7; j++)
                { // перевіряємо для неї всі суміжні вершини
                    if (mas[node, j] == 1 && nodes[j] == 0)
```

```
                    { // якщо вершина суміжна та не знайдена
                        queue.Enqueue(j); // додаємо її в чергу
                        nodes[j] = 1; // помічаємо вершину як
знайдену
                    }
                }
                Console.WriteLine(node + 1); // виводимо номер
вершини
            }
            Console.ReadLine();
        }
    }
}
```

Результат роботи програми:



Приклад 2.2.2. Написати програму для обходу графа в глибину.



Граф представлений матрицею суміжності:

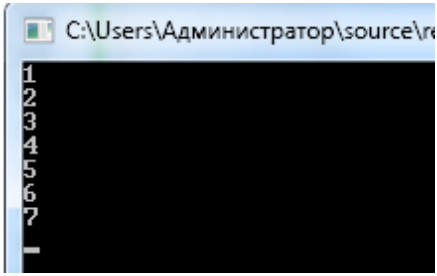
0	1	1	0	0	0	1
1	0	1	1	0	0	0
1	1	0	0	0	0	0
0	1	0	0	1	0	0
0	0	0	1	0	1	0
0	0	0	0	1	0	1
1	0	0	0	0	1	0

1) Нерекурсивний алгоритм:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp16
{
    class Program
    {
        static void Main(string[] args)
        {
            Stack<int> stack = new Stack<int>();
            // матрица суміжності
          int[,] mas = new int[7, 7] { { 0, 1, 1, 0, 0, 0, 1 },
                                       { 1, 0, 1, 1, 0, 0, 0 },
                                       { 1, 1, 0, 0, 0, 0, 0 },
                                       { 0, 1, 0, 0, 1, 0, 0 },
                                       { 0, 0, 0, 1, 0, 1, 0 },
                                       { 0, 0, 0, 0, 1, 0, 1 },
                                      { 1, 0, 0, 0, 0, 1, 0 } };
            int[] nodes = new int[7]; // вершини графа
            for (int i = 0; i < 7; i++) // на початку всі
вершини равны 0 (не відвідані)
                nodes[i] = 0;
            stack.Push(0); //розміщуємо в стек першу вершину
            while (stack.Count > 0)
            { // поки стек не порожній
                int node = stack.Pop(); // витягаємо вершину
                if (nodes[node] == 2) continue;
                nodes[node] = 2; // помічаємо її як відвідану
                for (int j = 6; j >= 0; j--)
                { // перевіряємо для неї всі суміжні вершини
                    if (mas[node, j] == 1 && nodes[j] != 2)
                    { // якщо вершина суміжна та не знайдена
                        stack.Push(j); // додаємо її  в стек
                        nodes[j] = 1; // помічаємо вершину як
знайдену
                    }
                }
                Console.WriteLine(node + 1); // виводимо номер
вершини
            }
```

```
            Console.ReadLine();
        }
    }
}
```

Результат роботи програми:

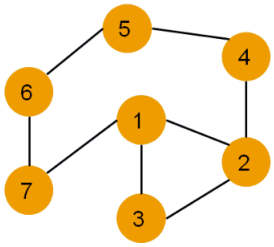


2) Рекрсивний алгоритм

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp18
{
    class Program
    {
    static int[,] mas = new int[7, 7] { { 0, 1, 1, 0, 0, 0, 1 },
                                        { 1, 0, 1, 1, 0, 0, 0 },
                                        { 1, 1, 0, 0, 0, 0, 0 },
                                        { 0, 1, 0, 0, 1, 0, 0 },
                                        { 0, 0, 0, 1, 0, 1, 0 },
                                        { 0, 0, 0, 0, 1, 0, 1 },
                                        { 1, 0, 0, 0, 0, 1, 0 }
};
        static int[] nodes = new int[7]; // вершини графа
        static void search(int st, int n)
        {
            int r;
            Console.Write((st + 1) + "    ");
            nodes[st] = 1;
```

```
            for (r = 0; r < n; r++)
                if ((mas[st,r] != 0) && (nodes[r] == 0))
                    search(r, n);
        }
        static void Main(string[] args)
        {
            for (int i = 0; i < 7; i++) // на початку всі
вершини равны 0 (не відвідані)
                nodes[i] = 0;
            search(0, 7);
            Console.ReadLine();
        }
    }
}
```

Приклад 2.2.3. Написати програму для знаходження найкоротшого шляху між першою та заданою вершинами графа.



Граф представлений матрицею суміжності:

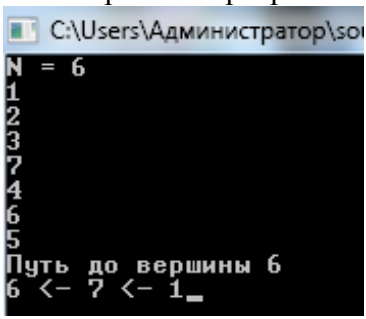
0	1	1	0	0	0	1
1	0	1	1	0	0	0
1	1	0	0	0	0	0
0	1	0	0	1	0	0
0	0	0	1	0	1	0
0	0	0	0	1	0	1
1	0	0	0	0	1	0

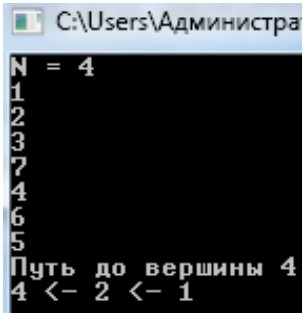
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp17
{
```

```
    class Program
    {
        public struct Edge
        {
            public int begin;
            public int end;
        };
        static void Main(string[] args)
        {
            Queue<int> queue = new Queue<int>();
            Stack<Edge> Edges = new Stack<Edge>();
            int req;
            Edge e;
           int[,] mas = new int[7, 7] { { 0, 1, 1, 0, 0, 0, 1 },
                                        { 1, 0, 1, 1, 0, 0, 0 },
                                        { 1, 1, 0, 0, 0, 0, 0 },
                                        { 0, 1, 0, 0, 1, 0, 0 },
                                        { 0, 0, 0, 1, 0, 1, 0 },
                                        { 0, 0, 0, 0, 1, 0, 1 },
                                        { 1, 0, 0, 0, 0, 1, 0 }
};

            int[] nodes = new int[7]; // вершини графа
            for (int i = 0; i < 7; i++) // исходно все вершины
равны 0
                nodes[i] = 0;
            Console.Write("N = ");
            req = Convert.ToInt32(Console.ReadLine());
            req--;
            queue.Enqueue(0); // розміщуємо в чергу першу
вершину
            while (queue.Count != 0)
            {
                int node = queue.Dequeue(); // витягаємо вершину
                nodes[node] = 2; // помічаємо її як відвідану
                for (int j = 0; j < 7; j++)
                {
                    if (mas[node, j] == 1 && nodes[j] == 0)
                    { // якщо вершина суміжна та не знайдена
                        queue.Enqueue(j); // додаємо її в чергу
                        nodes[j] = 1; // помічаємо вершину як
знайдену
                        e.begin = node;
                        e.end = j;
```

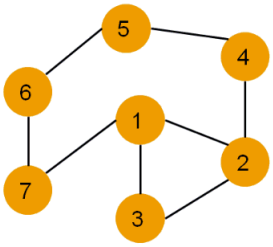
```
                        Edges.Push(e);
                        if (node == req) break;
                    }
                }
                Console.WriteLine(node + 1); // виводимо номер
вершини
            }
            Console.WriteLine("Путь до вершины " + (req + 1));
            Console.Write(req + 1);
            while (Edges.Count != 0)
            {
                e = Edges.Pop();
                if (e.end == req)
                {
                    req = e.begin;
                    Console.Write(" <- " + (req + 1));
                }
            }
            Console.ReadLine();
        }
    }
}
```

Результати роботи програми:





Приклад 2.2.4. Написати програму для знаходження лексикографічного першого шляху на графі.



Граф представлений матрицею суміжності:

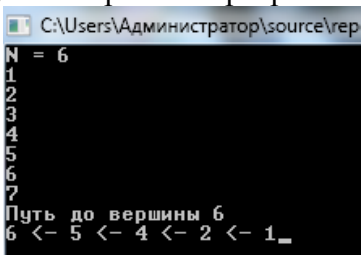
0	1	1	0	0	0	1
1	0	1	1	0	0	0
1	1	0	0	0	0	0
0	1	0	0	1	0	0
0	0	0	1	0	1	0
0	0	0	0	1	0	1
1	0	0	0	0	1	0

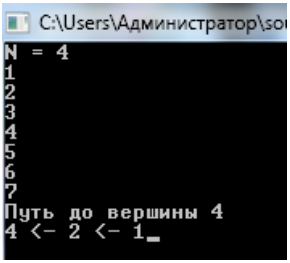
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp17
{
    class Program
    {
        public struct Edge
        {
            public int begin;
            public int end;
        };
        static void Main(string[] args)
        {
            Stack<int> stack = new Stack<int>();
            Stack<Edge> Edges = new Stack<Edge>();
            int req;
            Edge e;
           int[,] mas = new int[7, 7] { { 0, 1, 1, 0, 0, 0, 1 },
                                        { 1, 0, 1, 1, 0, 0, 0 },
                                        { 1, 1, 0, 0, 0, 0, 0 },
                                        { 0, 1, 0, 0, 1, 0, 0 },
                                        { 0, 0, 0, 1, 0, 1, 0 },
```

```
                                        { 0, 0, 0, 0, 1, 0, 1 },
                                        { 1, 0, 0, 0, 0, 1, 0 }
};

            int[] nodes = new int[7]; // вершини графа
            for (int i = 0; i < 7; i++) // исходно все вершины
равны 0
                nodes[i] = 0;
            Console.Write("N = ");
            req = Convert.ToInt32(Console.ReadLine());
            req--;
            stack.Push(0); // розміщуємо в чергу першу вершину
            while (stack.Count != 0)
            {
                int node = stack.Pop(); // витягаємо вершину
                if (nodes[node] == 2) continue;
                nodes[node] = 2; // помічаємо її як відвідану
                for (int j = 6; j >= 0; j--)
                {
                    if (mas[node, j] == 1 && nodes[j] != 2)
                    { // якщо вершина суміжна та не знайдена
                        stack.Push(j); // додаємо її в чергу
                        nodes[j] = 1; // помічаємо вершину як
знайдену
                        e.begin = node;
                        e.end = j;
                        Edges.Push(e);
                        if (node == req) break;
                    }
                }
                Console.WriteLine(node + 1); // виводимо номер
вершини
            }
            Console.WriteLine("Путь до вершины " + (req + 1));
            Console.Write(req + 1);
            while (Edges.Count != 0)
            {
                e = Edges.Pop();
                if (e.end == req)
                {
                    req = e.begin;
                    Console.Write(" <- " + (req + 1));
                }
            }
```

```
            Console.ReadLine();
        }
    }
}
```

Результати роботи програми:





2.3. Індивідуальні завдання

Потрібно реалізувати кожне завдання у відповідності з наведеними етапами:

1. вивчити словесну постановку задачі, виділяючи при цьому всі види даних;
2. сформулювати математичну постановку задачі;
3. обрати метод розв’язання задачі, якщо це необхідно;
4. розробити графічну схему алгоритму;
5. записати розроблений алгоритм мовою програмування;
6. розробити контрольний тест до програми;
7. налагодити програму;
8. представити звіт до роботи.

Варіант кожного завдання обирається за номером студента в журналі.

Завдання 1. Реалізувати програмне застосування (програму), яке виконує наступні функції:

1. Зчитати граф з вхідного файлу. На вхід подається текстовий файл наступного вигляду:

n m

v_1 u_1

v_2 u_2

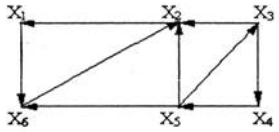
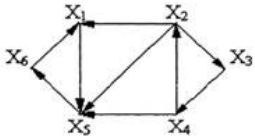
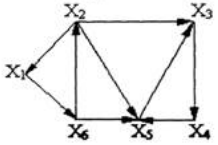
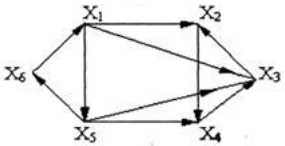
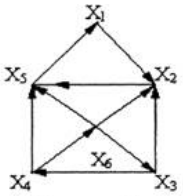
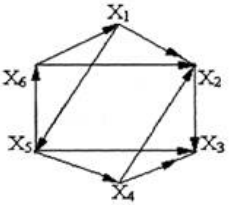
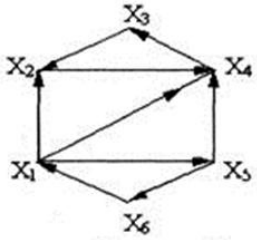
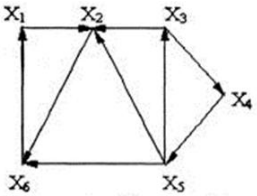
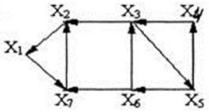
....

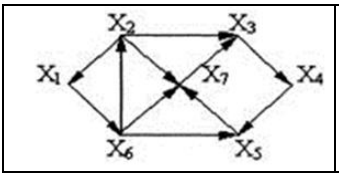
v_m u_m

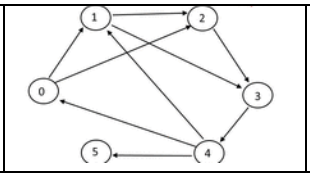
Тут n – кількість вершин графу (ціле число, більше нуля), m – кількість ребер графу (ціле число, більше нуля), v_i та u_i – початкова та кінцева вершина ребра i ($1 \le v_i \le n$, $1 \le u_i \le n$, цілі числа). Індексація вершин у файлі ведеться з 1. Вважається, що граф є орієнтованим. Таким чином можна сказати, що граф задається у файлі списком ребер.

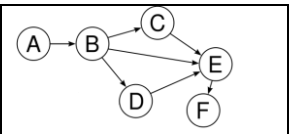
2. Вивести матриці інцидентності та суміжності. За вимогою користувача програма повинна виводити матриці інцидентності та суміжності (окремі функції) на екран та/або у текстовий файл, який вказує користувач.

Варіанти графів:

1 	2 	3 
4 	5 	6 
7 	8 	9 
10	11	12







Завдання 2.

Для графа з попереднього завдання реалізувати:

Варіант	Задача
1.	Обхід графа в ширину
2.	Обхід графа в глибину
3.	Знайти найкоротший шлях між першою та заданою вершинами графа
4.	Знайти лексикографічний перший шлях на графі
5.	Обхід графа в ширину
6.	Обхід графа в глибину
7.	Знайти найкоротший шлях між першою та заданою вершинами графа
8.	Знайти лексикографічний перший шлях на графі
9.	Обхід графа в ширину
10.	Обхід графа в глибину
11.	Знайти найкоротший шлях між першою та заданою вершинами графа
12.	Знайти лексикографічний перший шлях на графі

Завдання 3 (підвищеної складності).

1. Перевірити, чи є заданий неорієнтований граф зв'язним.
2. Циклом у графі називається маршрут, початкова і кінцева вершини якого збігаються. Перевірити, чи містить заданий неорієнтований граф хоча б один цикл.
3. Задано неорієнтований граф. Застосувавши алгоритм пошуку ушир, визначити всі вершини графу, відстань яких від заданої вершини s становить *d.*
4. Існує N міст. Для кожної пари міст *(i,j)* можна побудувати шлях, який з'єднає їх та не буде заходити до інших міст.

Вартість будівництва такого шляху стано вить a_{ij}. Визначити найдешевший спосіб будівництва шляхів, що дозволив би потрапити з кожного міста до будь-якого іншого.

5. Знайти найкоротший маршрут, що розпочинається і завершується в заданій вершині орієнтованого графу, проходячи через всі його вершини (якщо такий маршрут існує).
6. Заданий орієнтований граф з N вершинами. Обчислити кількість різних шляхів між усіма парами вершин графу.
7. Карта радіоактивного забруднення місцевості є прямокутною таблицею $N \times M$ у клітинках якої записані дані про рівень забруднення відповідної ділянки. Знайти шлях із лівої верхньої клітинки таблиці до правої нижньої, сумарна доза радіації на якому є найменшою. Ділянки шляху паралельні межам таблиці.
8. Застосувавши алгоритм пошуку вглибину, розробити програму пошуку в неорієнтованому зв'язаному графі шляху, який проходить один раз через кожне ребро в кожному напрямку.
9. Локальна мережа містить N комп'ютерів, окремі з яких заражені вірусом. Кожен канал зв'язку з'єднує певні два комп'ютери. Мережа вважається повністю ураженою, якщо жоден незаражений комп'ютер не з'єднаний із незараженим. Визначити мінімальну кількість комп'ютерів, зараження яких призведе до повного ураження мережі.
10. Певний механізм складається з деталей, які, у свою чергу, містять інші деталі. На виробництво кожної з них або на її складання з інших деталей витрачається певний час. Визначити загальний час, необхідний для виготовлення механізму, зобразивши послідовність складання деталей у вигляді орієнтованого графу.

2.4. Зміст звіту:

1. Прізвище та ім’я студента.
2. Номер і назва лабораторної роботи.
3. Мета роботи.
4. Номер індивідуального завдання. Текст завдання (постановка завдання).
5. Лістинг програми.
6. Скриншот робочої програми.
7. Висновок про засвоєнні знання та вміння.

2.5. Питання для самоперевірки

1. Дати визначення графу.
2. Як зображується граф в оперативній пам'яті?
3. Для яких задач кожен із способів зображення графу є найбільш ефективним?
4. Що таке матриця суміжності і матриця інциденцій?
5. Викладіть суть алгоритму Дейкстри.
6. Що таке обхід графу і для яких цілей він застосовується?
7. Поясніть, у чому полягає відмінність між алгоритмами обходу графу вглибину та вшир.
8. Якою є часова складність алгоритмів обходу графу вглибину та вшир?
9. Наведіть приклади задач, для розв'язання яких використовуються алгоритми обходу графу.

Лабораторна робота № 3. Хешування даних. Поняття хешування. Хеш-таблиці. Колізії. Алгоритми хешування. Відкрите і закрите хешування.

Мета роботи: Вивчити побудову функції хешування і алгоритмів хешування даних і навчитися розробляти алгоритми відкритого і закритого хешування при вирішенні задач на мові C#..

Послідовність виконання роботи:

1. Ознайомитись із теоретичними відомостями. (Актуалізація опорних знань).
2. Виконати програмування програм за поданими прикладами. Результат представити викладачу (*Застосування набутих знань*).
3. Виконати варіант самостійної роботи. (*Закріплення набутих знань*).
4. Оформити звіт на виконану роботу. (*Узагальнення та систематизація набутих знань*).
5. Захист звітів, відповіді на запитання.

3.1. Теоретичні відомості

Процес пошуку даних у великих обсягах інформації пов'язаний з часовими витратами, які обумовлені необхідністю перегляду та порівняння з ключем пошуку значного числа елементів. Скорочення пошуку можливо здійснити шляхом локалізації області перегляду. Наприклад, впорядкувати дані по ключу пошуку, розбити на блоки, які не перетинаються, за деякою груповою ознакою або поставити у відповідність реальним даним деякий код, що спростить процедуру пошуку.

В даний час використовується широко поширений метод забезпечення швидкого доступу до інформації, що зберігається в зовнішній пам'яті – хешування.

Хешування (англ. Hashing) – це перетворення вхідного масиву даних певного типу та довільної довжини в вихідний бітовий рядок фіксованої довжини. Такі перетворення також називаються хеш-функціями або функціями згортки, а їх результати називають хешем, хеш-кодом, хеш-таблицею або дайджестом повідомлення (англ. Message digest).

Хеш-таблиця – це структура даних, що реалізує інтерфейс асоціативного масиву, тобто вона дозволяє зберігати пари виду «ключ-значення» і виконувати три операції: операцію додавання нової пари, операцію пошуку і операцію видалення пари по ключу. Хеш-таблиця є масивом, який формується в певному порядку хеш-функцією.

Прийнято вважати, що хорошою, з точки зору практичного застосування, є така хеш-функція, яка задовольняє таким умовам:

- функція повинна бути простою з точки зору обчислювання;
- функція повинна розподіляти ключі в хеш-таблиці найбільш рівномірно;
- функція не повинна відображати будь-який зв'язок між значеннями ключів в зв'язок між значеннями адрес;
- функція повинна мінімізувати число колізій – тобто ситуацій, коли різним ключам відповідає одне значення хеш-функції (ключі в цьому випадку називаються синонімами).

При цьому перша властивість хорошої хеш-функції залежить від характеристик комп'ютера, а друге – від значень даних.

Якби всі дані були випадковими, то хеш-функції були б дуже прості (наприклад, кілька бітів ключа). Однак на практиці випадкові дані зустрічаються досить рідко, і доводиться створювати функцію, яка залежала б від усього ключа. Якщо хеш-функція розподіляє сукупність можливих ключів рівномірно по множині індексів, то хешування ефективно розбиває множину ключів. Найгірший випадок - коли всі ключі хешуються в один індекс.

При виникненні колізій необхідно знайти нове місце для зберігання ключів, які претендують на одну і ту ж комірку хеш-

таблиці. Причому, якщо колізії допускаються, то їх кількість необхідно мінімізувати. У деяких спеціальних випадках вдається уникнути колізій взагалі. Наприклад, якщо всі ключі елементів відомі заздалегідь (або дуже рідко змінюються), то для них можна знайти деяку ін'єкційну хеш-функцію, яка розподілить їх по осередках хеш-таблиці без колізій. Хеш-таблиці, що використовують подібні хеш-функції, не потребують механізмів вирішення колізій, і називаються хеш-таблицями з прямою адресацією.

Хеш-таблиці повинні відповідати наступним властивостям.

- Виконання операції в хеш-таблиці починається з обчислення хеш-функції від ключа. Отримане хеш-значення є індексом у вихідному масиві.
- Кількість збережених елементів масиву, поділене на число можливих значень хеш-функції, називається коефіцієнтом заповнення хеш-таблиці (load factor) і є важливим параметром, від якого залежить середній час виконання операцій.
- Операції пошуку, вставки і видалення повинні виконуватися в середньому за час O(1). Однак при такій оцінці не враховуються можливі апаратні витрати на перебудову індексу хеш-таблиці, пов'язану зі збільшенням значення розміру масиву і додаванням в хеш-таблицю нової пари.
- Механізм вирішення колізій є важливою складовою будь-якої хеш-таблиці.

Хешування корисно, коли широкий діапазон можливих значень повинен бути збережений в малому обсязі пам'яті, і потрібен спосіб швидкого, практично довільного доступу. Хеш-таблиці часто застосовуються в базах даних, і, особливо, в мовних процесорах типу компіляторів і ассемблеров, де вони підвищують швидкість обробки таблиці ідентифікаторів. Як використання хешування в повсякденному житті можна навести приклади розміщення книг в бібліотеці по тематичним каталогам, упорядкування в словниках за першими літерами

слів, шифрування спеціальностей у вищих навчальних закладах і т.д.

Методи вирішення колізій. *Колізії* ускладнюють використання хеш-таблиць, так як порушують однозначність відповідності між хеш-кодами і даними. Проте, існують способи подолання виникаючих складнощів:

- метод ланцюжків (зовнішнє або відкрите хешування);
- метод відкритої адресації (закрите хешування).

Метод ланцюжків. Технологія зчеплення елементів полягає в тому, що елементи множини, яким відповідає одне і те ж хеш-значення, зв'язуються в ланцюжок-список. У позиції номер «i» зберігається покажчик на голову списку тих елементів, у яких хеш-значення ключа дорівнює «i»; якщо таких елементів у множині немає, в позиції «i» записаний *null*. На рис. 3.1. демонструється реалізація методу ланцюжків при вирішенні колізій. На ключ 002 претендують два значення, які організовуються в лінійний список.

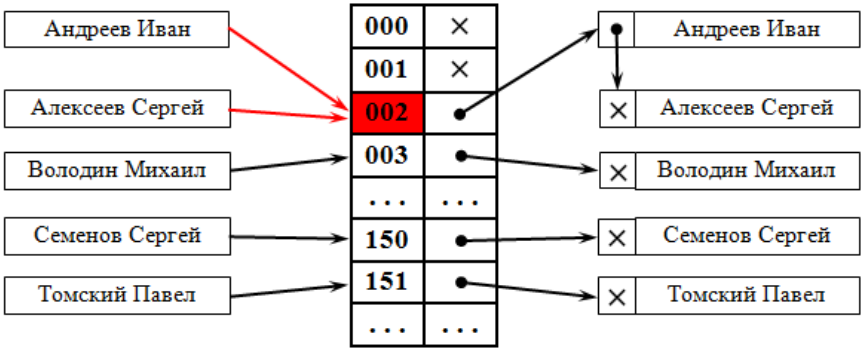


Рис. 3.1. Вирішення колізій за допомогою ланцюжків

Кожний осередок масиву є покажчиком на зв'язний список (ланцюжок) пар ключ-значення, що відповідають одному і тому ж хеш-значенню ключа. Колізії просто призводять до того, що з'являються ланцюжки довжиною більше одного елемента.

Операції пошуку або видалення даних вимагають перегляду всіх елементів відповідного йому ланцюжка, щоб знайти в ній елемент із заданим ключем. Для додавання даних потрібно

додати елемент в кінець або початок відповідного списку, і, в разі якщо коефіцієнт заповнення стане занадто великий, збільшити розмір масиву і перебудувати таблицю.

При припущенні, що кожен елемент може потрапити в будь-яку позицію таблиці з однаковою ймовірністю і незалежно від того, куди потрапив будь-який інший елемент, середній час роботи операції пошуку елемента становить O(1 + k), де k – коефіцієнт заповнення таблиці.

Метод відкритої адресації. На відміну від хешування з ланцюжками, при відкритої адресації ніяких списків немає, а всі записи зберігаються в самій хеш-таблиці. Кожна клітинка таблиці містить або елемент динамічної множини, або *null*.

В цьому випадку, якщо осередок з обчисленим індексом зайнята, то можна просто переглядати такі записи таблиці по порядку до тих пір, поки не буде знайдений ключ K або порожня позиція в таблиці. Для обчислення кроку можна також застосувати формулу, яка і визначить спосіб зміни кроку. На рис. 3.2 вирішення колізій здійснюється методом відкритої адресації. Два значення претендують на ключ 002, для одного з них знаходиться перше вільне (ще незайняте) місце в таблиці.

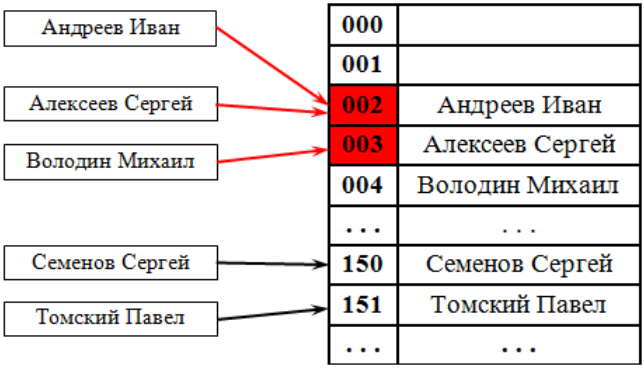


Рис. 3.2. Вирішення колізій за допомогою відкритої адресації

При будь-якому методі вирішення колізій необхідно обмежити довжину пошуку елемента. Якщо для пошуку елемента необхідно більше 3-4 порівнянь, то ефективність

використання такої хеш-таблиці пропадає і її слід реструктуризувати (тобто знайти іншу хеш-функцію), щоб мінімізувати кількість порівнянь для пошуку елемента

Для успішної роботи алгоритмів пошуку, послідовність проб повинна бути такою, щоб усі осередки хеш-таблиці опинилися переглянутими рівно по одному разу.

Видалення елементів в такій схемі утруднено. Зазвичай діють в такий спосіб: заводять логічний прапор для кожного осередку, щоб позначати, чи вилучений в ній елемент. Тоді видалення елемента полягає в установці цього прапора для відповідного осередку хеш-таблиці, але при цьому необхідно модифікувати процедуру пошуку існуючого елемента так, щоб вона вважала видалені осередки зайнятими, а процедуру оновлення - щоб вона їх вважала вільними і скидала значення прапора при додаванні.

Алгоритми хешування

Існує кілька типів функцій хешування, кожна з яких має свої переваги і недоліки і заснована на представленні даних. Наведемо огляд і аналіз деяких найбільш простих з застосовуваних на практиці хеш-функцій.

Таблиця прямого доступу

Найпростішою організацією таблиці, що забезпечує ідеально швидкий пошук, є таблиця прямого доступу. У такій таблиці ключ є адресою записи в таблиці або може бути перетворений на адресу, причому таким чином, що ніякі два різних ключа не перетворюються в один і той же адресу. При створенні таблиці виділяється пам'ять для зберігання всієї таблиці і заповнюється порожніми записами. Потім записи вносяться в таблицю - кожна на своє місце, яке визначається її ключем. При пошуку ключ використовується як адреса і за цією адресою вибирається запис. Якщо обрана запис порожня, то записи з таким ключем взагалі немає в таблиці. Таблиці прямого доступу дуже ефективні у використанні, але, на жаль, область їх застосування досить обмежена.

Назвемо простором ключів безліч всіх теоретично можливих значень ключів записи. Назвемо простором записів безліч тих

осередків пам'яті, які виділяються для зберігання таблиці. Таблиці прямого доступу застосовні тільки для таких завдань, в яких розмір простору записів може бути дорівнює розміру простору ключів. У більшості реальних задач розмір простору записів багато менше, ніж простору ключів. Так, якщо в якості ключа використовується прізвище, то, навіть обмеживши довжину ключа десятьма символами кирилиці, отримуємо 3310 можливих значень ключів. Навіть якщо ресурси обчислювальної системи і дозволять виділити простір записів такого розміру, то значна частина цього простору буде заповнена порожніми записами,

З метою економії пам'яті можна призначати розмір простору записів рівним розміру фактичного безлічі записів або перевершує його незначно. В цьому випадку необхідно мати деяку функцію, що забезпечує відображення точки з простору ключів в точку в просторі записів, тобто, перетворення ключа на адресу записи: *a=h(k)*, де *a* - адреса, *k*- ключ.

Ідеальною хеш-функцією є ін'єктивна функція, яка для будь-яких двох неоднакових ключів дає неоднакові адреси.

Метод залишків від ділення

Найпростішою хеш-функцією є ділення по модулю числового значення ключа Key на розмір простору записи *HashTableSize.* Результат інтерпретується як адреса записи. Слід мати на увазі, що така функція добре відповідає першому, але погано - останнім трьом вимогам до хеш-функції і сама по собі може бути застосована лише в дуже обмеженому діапазоні реальних завдань. Однак операція ділення по модулю зазвичай застосовується як останній крок в більш складних функціях хешування, забезпечуючи приведення результату до розміру простору записів.

Якщо ключів менше, ніж елементів масиву, то в якості хеш-функції можна використовувати ділення по модулю, тобто залишок від ділення цілочисельного ключа *Key* на розмірність масиву *HashTableSize*, тобто:

Key % HashTableSize

Ця функція дуже проста, хоча і не відноситься до хороших. Взагалі, можна використовувати будь-яку розмірність масиву, але вона повинна бути такою, щоб мінімізувати число колізій. Для цього в якості розмірності краще використовувати просте число. У більшості випадків подібний вибір цілком задовільний. Для символьного рядка ключем може бути залишок від ділення, наприклад, суми кодів символів рядка на *HashTableSize*.

На практиці, метод ділення - найпоширеніший.

```
// функція створення хеш-таблиці метод поділу по модулю
int Hash (int Key, int HashTableSize) {
// HashTableSize
 return Key %  HashTableSize;
}
```

Метод функції середини квадрата

Наступною хеш-функцією є функція середини квадрата. Значення ключа перетворюється в число, це число потім зводиться до квадрату, з нього вибираються кілька середніх цифр і інтерпретуються як адреса записи.

Метод згортки

Ще однією хеш-функцією можна назвати функцію згортки. Цифрове подання ключа розбивається на частини, кожна з яких має довжину, рівну довжині необхідного адреси. Над частинами здійснюються певні арифметичні або порозрядні логічні операції, результат яких інтерпретується як адреса. Наприклад, для порівняно невеликих таблиць з ключами-символьними рядками непогані результати дає функція хешування, в якій адреса записи виходить в результаті додавання кодів символів, що складають рядок-ключ.

Як хеш-функції також застосовують функцію перетворення системи числення. Ключ, записаний як число в деякій системі числення P, інтерпретується як число в системі числення $Q > P$. Зазвичай вибирають $Q = P + 1$. Це число перекладається з системи Q назад в систему P, приводиться до розміру простору записів і інтерпретується як адреса.

Відкрите хешування

Основна ідея базової структури при відкритому (зовнішньому) хешуванні полягає в тому, що потенційна множина (можливо, нескінченна) розбивається на кінцеве число класів. Для B класів, пронумерованих від 0 до B-1, будується хеш-функція h(x) така, що для будь-якого елемента x вихідної множини функція h(x) набуває цілочисельне значення з інтервалу 0,1, ..., B-1, що відповідає класу, якому належить елемент x. Часто класи називають сегментами, тому будемо говорити, що елемент x належить сегменту h(x). Масив, званий таблицею сегментів і проіндексований номерами сегментів 0,1, ..., B-1, містить заголовки для B списків. Елемент x, що відноситься до i-го списку - це елемент вихідної множини, для якого h (x) = i.

Якщо сегменти приблизно однакові за розміром, то в цьому випадку списки всіх сегментів повинні бути найбільш короткими при даному числі сегментів. Якщо вихідна безліч складається з N елементів, тоді середня довжина списків буде N/B елементів. Якщо можна оцінити величину N і вибрати B якомога ближче до цієї величини, то в кожному списку буде один або два елементи. Тоді час виконання операторів словників буде малою постійною величиною, що не залежить від N.

Закрите хешування. При закритому (внутрішньому) хешуванні в хеш-таблиці зберігаються безпосередньо самі елементи, а не заголовки списків елементів. Тому в кожного запису (сегменті) може зберігатися тільки один елемент. При закритому хешуванні застосовується методика повторного хешування. Якщо здійснюється спроба помістити елемент x в сегмент з номером h(x), який вже зайнятий іншим елементом (колізія), то відповідно до методики повторного хешування вибирається послідовність інших номерів сегментів h1(x), h2(x), ..., куди можна помістити елемент x. Кожне з цих місць розташування послідовно перевіряється, поки не буде знайдено вільне. Якщо вільних сегментів немає, то, отже, таблиця заповнена, і елемент x додати не можна.

При пошуку елемента x необхідно переглянути всі місцеположення h(x), h1(x), h2(x), ..., поки не буде знайдений x або поки не зустрінеться порожній сегмент. Щоб пояснити, чому можна зупинити пошук при досягненні порожнього сегмента, припустимо, що в хеш-таблиці не допускається видалення елементів. Нехай h3(x) - перший порожній сегмент. У такій ситуації неможливо знаходження елемента x в сегментах h4(x), h5(x)і далі, так як при вставці елемент x вставляється в перший порожній сегмент, отже, він знаходиться десь до сегмента h3(x). Але якщо в хеш-таблиці допускається видалення елементів, то при досягненні порожнього сегмента, не знайшовши елементу x, не можна бути впевненим у тому, що його взагалі немає в таблиці, так як сегмент може стати порожнім уже після вставки елементах. Тому, щоб збільшити ефективність даної реалізації, необхідно в сегмент, який звільнився після операції видалення елемента, помістити спеціальну константу, яку назвемо, наприклад, *DEL*. В якості альтернативи спеціальної константі можна використовувати додаткове поле таблиці, яке показує стан елемента. Важливо розрізняти константи *DEL* і NULL - остання знаходиться в сегментах, які ніколи не містили елементів. При такому підході виконання пошуку елемента не вимагає перегляду всієї хеш-таблиці. Крім того, при вставці елементів сегменти, помічені константою *DEL*, Можна трактувати як вільні, таким чином, простір, звільнене після видалення елементів, можна рано чи пізно використовувати повторно. Але якщо неможливо безпосередньо відразу після видалення елементів помітити що звільнилися сегменти, то слід віддати перевагу закритому хешування схему відкритого хешування.

Існує кілька методів повторного хешування, тобто визначення місць розташування h(x), h1(x), h2(x), ...:

- лінійне випробування;
- квадратичное опробование;
- двойное хеширование.

Лінійне випробування зводиться до послідовного перебору сегментів таблиці з деяким фіксованим кроком:

$$адреса = h(x) + ci,$$

де *i*- номер спроби розв'язати колізію;
c- константа, яка визначає крок перебору.

При кроці, що дорівнює одиниці, відбувається послідовний перебір всіх сегментів після поточного. *Квадратичне випробування* відрізняється від лінійного тим, що крок перебору сегментів нелінійно залежить від номера спроби знайти вільний сегмент:

$$адреса = h(x) + ci + di2,$$

де *i*- номер спроби дозволити колізію,
c і *d* - константи.

Завдяки нелінійності такої адресації зменшується число проб при великому числі ключів-синонімів. Однак навіть відносно невелике число проб може швидко призвести до виходу за адресний простір невеликої таблиці внаслідок квадратичної залежності адреси від номера спроби.

Ще один різновид методу відкритої адресації, яка називається подвійним хешування, заснована на нелінійній адресації, що досягається за рахунок підсумовування значень основної і додаткової хеш-функцій:

$$адреса = h(x) + ih2(x).$$

Очевидно, що по мірі заповнення хеш-таблиці будуть відбуватися колізії, і в результаті їх вирішення чергова адреса може вийти за межі адресного простору таблиці. Щоб це явище відбувалося рідше, можна піти на збільшення довжини таблиці у порівнянні з діапазоном адрес, що видаються хеш-функцією. З одного боку, це призведе до скорочення числа колізій і прискорення роботи з хеш-таблицею, а з іншого – нераціональне витрачання пам'яті. Навіть при збільшенні довжини таблиці в два рази в порівнянні з областю значень хеш-функції немає гарантії того, що в результаті колізій адреса не перевищить довжину таблиці. При цьому в початковій частині таблиці може

залишатися достатньо вільних сегментів. Тому на практиці використовують циклічний перехід до початк таблиці.

Однак у випадку багаторазового перевищення адресного простору і, відповідно, багаторазового циклічного переходу до початку відбуватиметься перегляд одних і тих же раніше зайнятих сегментів, тоді як між ними можуть бути ще вільні сегменти. Більш коректним буде використання зсуву адреси на 1 у випадку кожного циклічного переходу до початку таблиці. Це підвищує вірогідність знаходження вільних сегментів.

У разі застосування схеми закритого хешування швидкість виконання вставки та інших операцій залежить не тільки від рівномірності розподілу елементів за сегментами хеш-функцією, але і від обраної методики повторного хешування (випробування) для вирішення колізій, пов'язаних із спробами вставки елементів у вже заповнені сегменти. Наприклад, методика лінійного випробування для вирішення колізій – не найкращий вибір. Як тільки кілька послідовних сегментів будуть заповнені, утворюючи групу, будь-який новий елемент при спробі вставки в ці сегменти буде вставлений в кінець цієї групи, збільшуючи тим самим довжину групи послідовно заповнених сегментів. Іншими словами, для пошуку порожнього сегмента в разі безперервного розташування заповнених сегментів необхідно переглянути більше сегментів, ніж при випадковому розподілі заповнених сегментів. Звідси також випливає очевидний висновок, що при безперервному розташуванні заповнених сегментів збільшується час виконання вставки нового елемента і інших операцій.

Ідея хешування вперше була висловлена Г. П. Ланом при створенні внутрішнього меморандуму IBM в січні 1953 року з пропозицією використовувати для вирішення колізій метод ланцюжків. Приблизно в цей же час інший співробітник IBM, Жіні Амдал, висловила ідею використання відкритої лінійної адресації. У відкритій пресі хешування вперше було описано Арнольдом Думи (1956 рік), який вказав, що в якості хеш-адреси зручно використовувати залишок від ділення на просте

число. А. Думи описував метод ланцюжків для вирішення колізій, але не говорив про відкриту адресацію. Підхід до хешування, відмінний від методу ланцюжків, був запропонований А. П. Єршовим (1957 рік), який розробив і описав метод лінійної відкритої адресації.

Ключові терміни

Вторинні ключі – це ключі, які не дозволяють однозначно ідентифікувати запис у таблиці.

Закрите хешування або Метод відкритої адресації – це технологія вирішення колізій, яка передбачає зберігання записів в самій хеш-таблиці.

Колізія – це ситуація, коли різним джерелам відповідає одне значення хеш-функції.

Коефіцієнт заповнення хеш-таблиці – це кількість збережених елементів масиву, поділена на число можливих значень хеш-функції.

Відкрите хешування або Метод ланцюжків – це технологія вирішення колізій, яка полягає в тому, що елементи множини з рівними хеш-значеннями зв'язуються в ланцюжок-список.

Первинні ключі – це ключі, дозволяють однозначно ідентифікувати запис.

Повторне хешування – це пошук місця для чергового елемента таблиці з урахуванням кроку переміщення.

Простір записів – це множина тих елементів пам'яті, які виділяються для зберігання таблиці.

Простір ключів – це безліч всіх теоретично можливих значень ключів запису.

Синоніми – це ключі збігаються в хеш-таблиці.

Хешування – це перетворення вхідного масиву даних певного типу і довільної довжини в вихідний бітову рядок фіксованої довжини.

Хеш-таблиця – це структура даних, що реалізує інтерфейс асоціативного масиву, тобто вона дозволяє зберігати пари виду «ключ – значення» і виконувати три операції: операцію

додавання нової пари, операцію пошуку і операцію видалення пари по ключу.

Хеш-таблиці з прямою адресацією – це хеш-таблиці, які використовують инъективные хеш-функції і не потребують механізм вирішення колізій.

Короткі підсумки

1. В даний час використовується широко поширений метод забезпечення швидкого доступу до великих обсягів інформації - хешування.
2. Для встановлення відповідності ключів і даних будується хеш-таблиця.
3. Хеш-таблиця будується за допомогою хеш-функцій. Практичне застосування отримали функції прямого доступу, залишків від ділення, середини квадрата, згортки.
4. При побудові хеш-таблиць можуть виникати колізії, тобто ситуації неоднозначного відповідності даних ключу.
5. Дозвіл колізій проводиться методом ланцюжків (відкрите або зовнішнє хешування) або методом відкритої адресації (закрите хешування).
6. Пошук вільних ключів в методі відкритої адресації може проводитися методом повторного хешування за допомогою лінійного випробування, квадратичного випробування або подвійного хешування.
7. Ідентифікація даних в таблицях може здійснюватися як з первинного, так і по вторинному ключу.
8. Хешування має широке практичне застосування в теорії баз даних, кодування, банківській справі, криптографії та інших областях.

3.2. Приклади виконання завдань

Приклад 3.2.1 Програмна реалізація відкритого хешування.

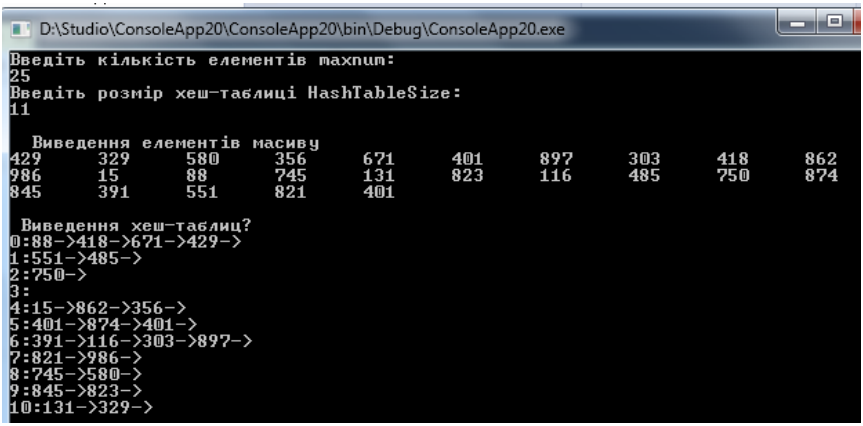
```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;


namespace ConsoleApp20
{
    using T = System.Int32; //створення синоніму типа
    using hashTableIndex = System.Int32; //створення синоніму
типа

    class Node
    {
        public T data; // дані, що зберігаються в вершині
        public Node next; // наступна вершина
    }
    class Program
    {
        static Node[] hashTable;
        static T hashTableSize;
        // хеш-функція розміщення вершини
        static hashTableIndex myhash(T data)
        {
            return (data % hashTableSize);
        }
        // функція пошуку місця розташування і вставки вершини в
таблицю
        static Node insertNode(T data)
        {
            Node  p,  p0;
            hashTableIndex bucket;
            // вставка вершини в початок списку
            bucket = myhash(data);
            if ((p = new Node()) == null)
            {
                Console.WriteLine("Брак пам'яті (insertNode)
\n");
                return null;
            }
            p0 = hashTable[bucket];
            hashTable[bucket] = p;
            p.next = p0;
            p.data = data;
            return p;
```

```
        }
        // функція видалення вершини з таблиці
        static void deleteNode(T data)
        {
            Node p0, p;
            hashTableIndex bucket;
            p0 = null;
            bucket = myhash(data);
            p = hashTable[bucket];
            while (p!=null && !(p.data == data))
            {
                p0 = p;
                p = p.next;
            }
            if (p==null) return;
            if (p0!=null)
                p0.next = p.next;
            else
                hashTable[bucket] = p.next;
        }
        // функція пошуку вершини зі значенням data
        static  Node findNode(T data)
        {
            Node p;
            p = hashTable[myhash(data)];
            while (p != null && !(p.data == data))
                p = p.next;
            return p;
        }
        static void Main(string[] args)
        {
            int i,  maxnum;
            int[] a;
            Console.WriteLine("Введіть кількість елементів
maxnum:");
            maxnum=Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Введіть розмір хеш-таблиці
HashTableSize:");
            hashTableSize = Convert.ToInt32(Console.ReadLine());
            a = new int[maxnum];
            hashTable = new Node[hashTableSize];
            Random x = new Random();
            for (i = 0; i < hashTableSize; i++)
                hashTable[i] = null;
```

```
            // генерація масиву
            Random numb = new Random();
            for (i = 0; i < maxnum; i++)
                a[i] = numb.Next(1000);
            // заповнення хеш-таблиці елементами масиву
            for (i = 0; i < maxnum; i++)
            {
                insertNode(a[i]);
            }
            // пошук елементів масиву по хеш-таблиці
            for (i = maxnum - 1; i >= 0; i--)
            {
                findNode(a[i]);
            }
            // Виведення елементів масиву на екран
            Console.WriteLine("\n  Виведення елементів масиву");
            for (i = 0; i < maxnum; i++)
            {
                Console.Write( a[i]);
                if (i < maxnum - 1) Console.Write( "\t");
            }
            // Виведення хеш-таблиці на екран
            Console.WriteLine("\n\n Виведення хеш-таблиці");
            for (i = 0; i < hashTableSize; i++)
            {
                Console.Write( i + ":");
                Node Temp = hashTable[i];
                while (Temp!=null)
                {
                    Console.Write( Temp.data + "->");
                    Temp = Temp.next;
                }
                Console.WriteLine();
            }
            Console.ReadLine();
        }
    }
}
```

Результат роботи програми:



Завдання 3.2.2. Програмна реалізація закритого хешування.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;


namespace ConsoleApp20
{
    using T = System.Int32;
    using hashTableIndex = System.Int32;


    class Program
    {
        static T[] hashTable;
        static T hashTableSize;
        static bool[] used;
        // хеш-функція розміщення вершини

        static hashTableIndex myhash(T data)
        {
            return (data % hashTableSize);
        }

        // функция вычисления расстояние от a до b (по часовой
стрелке, слева направо)
```

```
        static int dist(hashTableIndex a, hashTableIndex b)
        {
            return (b - a + hashTableSize) % hashTableSize;
        }
        // функція пошуку місця розташування і вставки величини
в таблицю
        static void insertData(T data)
        {
            hashTableIndex bucket;
            bucket = myhash(data);
            while (used[bucket] && hashTable[bucket] != data)
                bucket = (bucket + 1) % hashTableSize;
            if (!used[bucket])
            {
                used[bucket] = true;
                hashTable[bucket] = data;
            }
        }

        // функція видалення вершини з таблиці
        static void deleteData(T data)
        {
            {
                int bucket, gap;
                bucket = myhash(data);
                while (used[bucket] && hashTable[bucket] !=
data)
                    bucket = (bucket + 1) % hashTableSize;
                if (used[bucket] && hashTable[bucket] == data)
                {
                    used[bucket] = false;
                    gap = bucket;
                    bucket = (bucket + 1) % hashTableSize;
                    while (used[bucket])
                    {
                        if (bucket == myhash(hashTable[bucket]))
                            bucket = (bucket + 1) %
hashTableSize;
                        else if (dist(myhash(hashTable[bucket]),
bucket) < dist(gap, bucket))
                            bucket = (bucket + 1) %
hashTableSize;
                        else
                        {
```

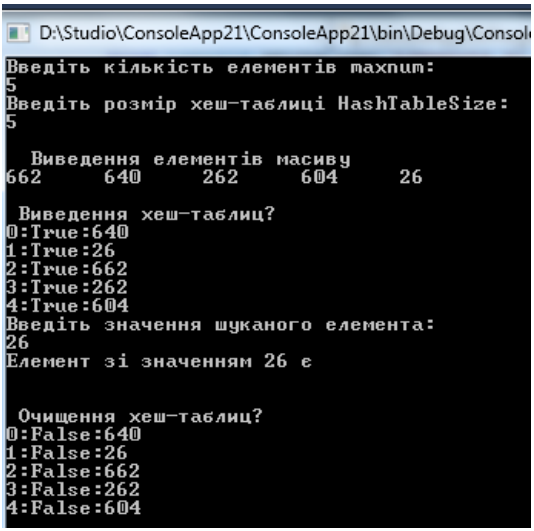
```
                        used[gap] = true;
                        hashTable[gap] = hashTable[bucket];
                        used[bucket] = false;
                        gap = bucket;
                        bucket++;
                    }
                }
            }
        }

        // функция поиска величины, равной data
        static bool findData(T data)
        {
            hashTableIndex bucket;
            bucket = myhash(data);
            while (used[bucket] && hashTable[bucket] != data)
                bucket = (bucket + 1) % hashTableSize;
            return used[bucket] && hashTable[bucket] == data;
        }


        static void Main(string[] args)
        {
            int i, maxnum;
            int[] a;
            Console.WriteLine("Введіть кількість елементів
maxnum:");
            maxnum = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Введіть розмір хеш-таблиці
HashTableSize:");
            hashTableSize = Convert.ToInt32(Console.ReadLine());
            a = new int[maxnum];
            hashTable = new T[hashTableSize];
            used = new bool[hashTableSize];
            // ініціалізація  хеш-таблиці та масиву заповненості
            for (i = 0; i < hashTableSize; i++)
            {
                hashTable[i] = 0;
                used[i] = false;
            }
            // генерація масиву
            Random numb = new Random();
            for (i = 0; i < maxnum; i++)
```

```
                a[i] = numb.Next(1000);
            // заповнення хеш-таблиці елементами масиву
            for (i = 0; i < maxnum; i++)
            {
                insertData(a[i]);
            }
            // Виведення елементів масиву на екран
            Console.WriteLine("\n  Виведення елементів масиву");
            for (i = 0; i < maxnum; i++)
            {
                Console.Write(a[i]);
                if (i < maxnum - 1) Console.Write("\t");
            }
            // Виведення хеш-таблиці на екран
            Console.WriteLine("\n\n Виведення хеш-таблиці");
            for (i = 0; i < hashTableSize; i++)
            {
                Console.WriteLine ( i + ":" + used[i] + ":" +
hashTable[i]);
            }

            // пошук елементів масиву по хеш-таблиці
            Console.WriteLine("Введіть значення шуканого
елемента:");
            T N = Convert.ToInt32(Console.ReadLine());
            if (findData(N)) Console.WriteLine("Елемент зі
значенням " + N + " є");
            else Console.WriteLine("Елемента зі значенням " + N
+ " немає");

            // очищення хеш-таблиці
            for (i = maxnum - 1; i >= 0; i--)
            {
                deleteData(a[i]);
            }
            Console.WriteLine("\n\n Виведення хеш-таблиці");
            for (i = 0; i < hashTableSize; i++)
            {
                Console.WriteLine(i + ":" + used[i] + ":" +
hashTable[i]);
            }
            Console.ReadLine();
        }
    }
```

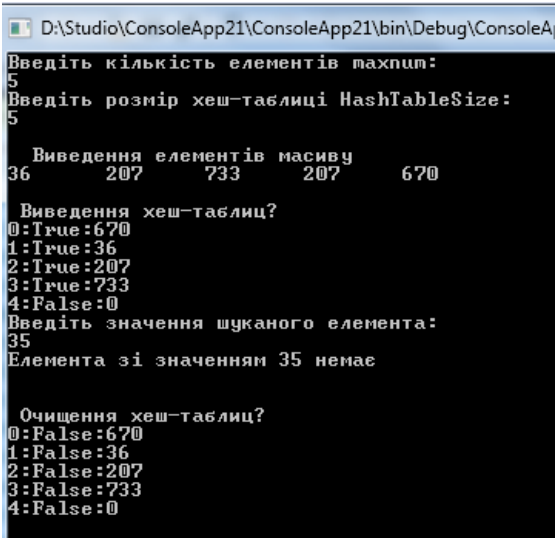
}

Результати роботи програми

1)



2)



3.3. Індивідуальні завдання

Потрібно реалізувати кожне завдання у відповідності з наведеними етапами:

1. вивчити словесну постановку задачі, виділяючи при цьому всі види даних;
2. сформулювати математичну постановку задачі;
3. обрати метод розв'язання задачі, якщо це необхідно;
4. розробити графічну схему алгоритму;
5. записати розроблений алгоритм мовою програмування;
6. розробити контрольний тест до програми;
7. налагодити програму;
8. представити звіт до роботи.

Варіант кожного завдання обирається за номером студента в журналі.

При виконанні лабораторної роботи для кожного завдання потрібно написати програму на мові C#, яка отримує дані з клавіатури або з вхідного файлу, виконує їх обробку відповідно до вимог завдання і виводить результат у вихідний файл. Для обробки даних необхідно реалізувати функції алгоритмів хешування даних. Обмеженнями на вхідні дані є максимальний розмір строкових даних, допустимий діапазон значень використовуваних числових типів в мові C#/

Завдання 1. Створіть хеш-таблицю з елементів вашого варіанту. Хеш-функція повинна прагнути до оптимальної. Визначте найкращий і найгірший варіант розподілу елементів.

№ варіанту	Завдання	Метод хешування
1.	Шестизначні номера пристроїв у вигляді 6xx1xx	Закрите хешування
2.	Слова російської мови	Відкрите хешування
3.	Слова англійської мови	Закрите

		хешування
4.	ПІБ співробітників фірми	Відкрите хешування
5.	Цілі числа	Закрите хешування
6.	Координати точки на площині у вигляді (x, y)	Відкрите хешування
7.	Номери телефонів в десятизначним форматі +x (xxx) xxx-xx-xx	Закрите хешування
8.	Математичні терміни	Відкрите хешування
9.	ІПН у вигляді 77xxxxx54x	Закрите хешування
10.	Номери документів у вигляді 3ВехххххААх	Відкрите хешування
11.	Номери автомобілів.	Закрите хешування
12.	Назви картин	Відкрите хешування
13.	Електронна адреса виду http://www.xxxxxxx.ua	Закрите хешування
14.	Поштова адреса виду xxxxxxxx@gmail.com	Відкрите хешування

Завдання 2 (підвищеної складності). Кожне завдання необхідно розв'язати у відповідності з вивченими алгоритмами хешування даних. Програму для вирішення кожного завдання необхідно розробити методом процедурної абстракції, використовуючи функції, коди яких потрібно супроводжувати коментарями. Результати обробки даних потрібно виводити у вихідний файл та дублювати виведення на екран.

1. Скласти хеш-таблицю, яка містить букви та кількість їх входження в уведеному рядку. Вивести таблицю на екран. Здійснити пошук введеної букви в хеш-таблиці.

2. Побудувати хеш-таблицю зі слів довільного текстового файлу, задаючи її розмірність з екрану. Вивести побудовану таблицю слів на екран. Здійснити пошук введеного слова. Виконати програму для різних розмірностей таблиці та порівняти кількість порівнянь. Видалити всі слова, які починаються на вказану букву, вивести таблицю.
3. Побудувати хеш-таблицю для зарезервованих слів певної мови програмування (не менше 20 слів), яка містить HELP для кожного слова. Вивести на екран підказку по введеному слову. Додати підказку для нового введеного слова, використовуючи за необхідності реструктуризацію таблиці. Порівняти ефективність додавання ключа в таблицю або її реструктуризацію для різної степені наповненості таблиці.
4. В текстовому файлі містяться цілі числа. Побудувати хеш-таблицю з чисел файлу. Здійснити пошук введеного цілого числа в хеш-таблиці. Порівняти результати кількості порівнянь для різних наборів даних в файлі.

3.4. Зміст|вміст| звіту:

1. Прізвище та ім'я студента.
2. Номер і назва лабораторної роботи.
3. Мета роботи.
4. Номер індивідуального завдання. Текст завдання (постановка завдання).
5. Лістинг програми.
6. Скриншот робочої програми.
7. Висновок про засвоєнні знання та вміння.

3.5. Питання для самоперевірки

1. Який принцип побудови хеш-таблиці?
2. Чи існують універсальні методи побудови хеш-таблиці? Відповідь обґрунтуйте.
3. Чому можливе виникнення колізій?

4. Які є методи розв’язання колізій? Схарактеризуйте їх ефективність в різних ситуаціях.
5. Назвіть переваги відкритого і закритого хешування.
6. В якому випадку пошук в хеш-таблицях стає неефективним?
7. Як обирається метод змінення адреси при повторному хешуванні?

Рекомендована література

1. Альфред Ахо. Структуры данных и алгоритмы. М. : Вильямс, 2007. 400 с.
2. Вирт Н. Алгоритмы и структуры данных. М., 2012. 272 с.
3. Седжвик Н. Фундаментальные алгоритмы на С++. Части 1-4.
4. Диасофт, 2001. 688 с.
5. Кнут Д. Искусство программирования. Т. 1-4. М.: Вильямс, 2006. 682 с.
6. Алгоритми і структури даних: конспект лекцій. Частина 1. Структури даних / Укладачі: О. Д. Воробйова, Л. В. Глазунова. Одеса : ОНАЗ ім. О. С. Попова, 2017. 48 с.