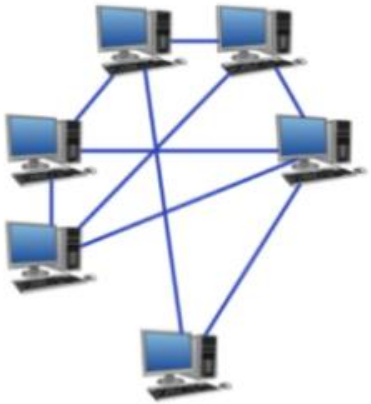
НЕЛІНІЙНІ СТРУКТУРИ ДАНИХ

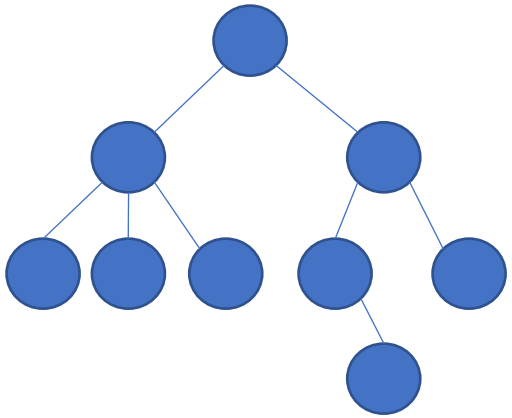
Лекція 6

Дисципліна «Алгоритми та структури даних»

Поняття нелінійної структури

Нелінійні структури відрізняються від лінійних тим, що вони не зберігають елементи послідовно (один за одним). Замість цього вони моделюють складні зв'язки





Граф (Graph)

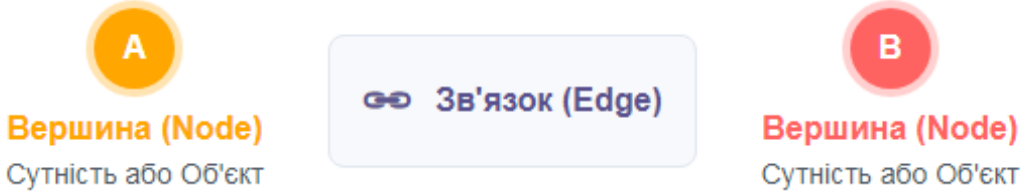
Дерево (Tree)

Хеш-таблиця (Hash Table / Map / Dictionary)

Застосування графів в IT

Теорія графів є математичною мовою для опису зв'язків. У світі даних це єдина структура, що здатна ефективно моделювати складні взаємозалежності - від соціальних мереж до маршрутизації пакетів

Граф складається з двох основних елементів: вершин (Vertices /Nodes), що представляють сутності, та ребер (Edges), що показують зв'язки між ними. Це базова модель для всіх складних IT-систем



Соціальні мережі та рекомендації

Сфера застосування: Моделювання відносин між користувачами, виявлення спільнот та впливових вузлів.

Приклад:

- **Вершини:** Користувачі.
- **Ребра:** Друзі, підписки, взаємодії.
- **Алгоритм:** Алгоритм пошуку шляху для рекомендації "друзів друзів" або для персоналізації стрічки новин.

Мережева маршрутизація (Routing)

Сфера застосування: Знаходження найшвидшого або найдешевшого шляху для передачі пакетів даних в інтернеті чи локальних мережах.

Приклад:

- **Вершини:** Маршрутизатори та Хости.
- **Ребра:** Мережеві з'єднання (кабелі, канали).
- **Алгоритм:** Алгоритм Дейкстри (Dijkstra's) або Bellman-Ford для визначення найкоротшого шляху з найменшою затримкою.

Застосування графів в IT

Веб-пошук та ранжування (SEO)

Сфера застосування: Оцінка важливості та авторитетності веб-сторінок на основі гіперпосилань.

Приклад:

- **Вершини:** Веб-сторінки.
- **Ребра:** Гіперпосилання (спрямовані).
- **Алгоритм:** PageRank — використовує структуру графа для визначення того, які сторінки є більш "авторитетними" і повинні відображатися вище у результатах пошуку.

Компілятори та аналіз коду

Сфера застосування: Оптимізація коду, виявлення залежностей між функціями або модулями, та управління пам'яттю.

Приклад:

- **Вершини:** Функції, змінні, блоки коду.
- **Ребра:** Виклик однієї функції іншою (Call Graph) або потік даних.
- **Алгоритм:** Виявлення циклічних залежностей у коді для запобігання нескінченним рекурсіям або проблем з компіляцією.

Графові бази даних (Graph DB)

Сфера застосування: Спеціалізовані NoSQL бази даних, які ефективно зберігають та обробляють сильно зв'язані дані.

Приклад:

- **Система:** Neo4j, JanusGraph.
- **Модель:** Пряме відображення даних на структуру графа.
- **Запит:** Знайти всіх людей, які знаходяться "за два рукостискання" від мене і працюють у тій же компанії.

Штучний інтелект

Сфера застосування: Побудова Knowledge Graphs (Графів Знань) для надання контексту та логічних зв'язків.

Приклад:

- **Вершини:** Об'єкти, поняття (Київ, столиця, Україна).
- **Ребра:** Відношення (Київ -> є столицею -> України).
- **Алгоритм:** Семантичний пошук та логічне виведення (Deductive Reasoning) для відповіді на складні запитання.

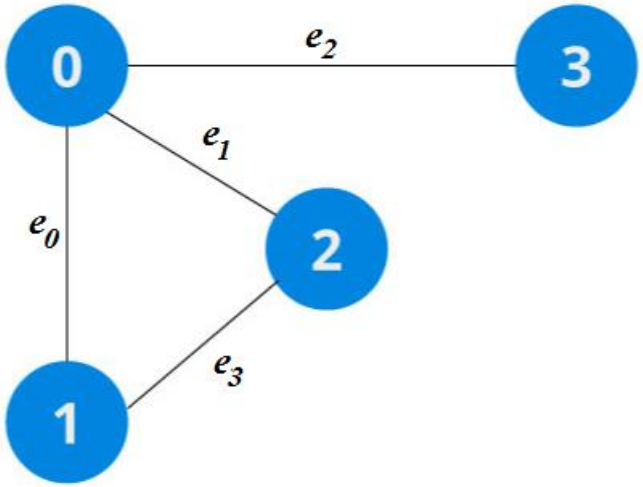
Граф

Граф (G) - це фундаментальна нелінійна структура даних, що використовується для моделювання зв'язків (відношень) між об'єктами

Формально граф G визначається як пара множин (V, E), де:

V **(Vertices)**: Множина **вершин** (вузлів, точок)

E **(Edges)**: Множина **ребер** (дуг, ліній), які з'єднують пари вершин



V = {0, 1, 2, 3} від англ. ***vertex***
E = {e_0, e_1, e_2, e_3,} від англ. ***edge***
e_0= (0, 1)
e_1= (0,2)
e_2= (0,3)
e_3= (1,2)
G = {V, E}

Основна термінологія

Вершина (Vertex) - основний елемент графа, що може містити дані

Ребро (Edge) - зв'язок між двома вершинами

Суміжні вершини (Adjacent Vertices) - дві вершини називаються суміжними, якщо вони з'єднані ребром

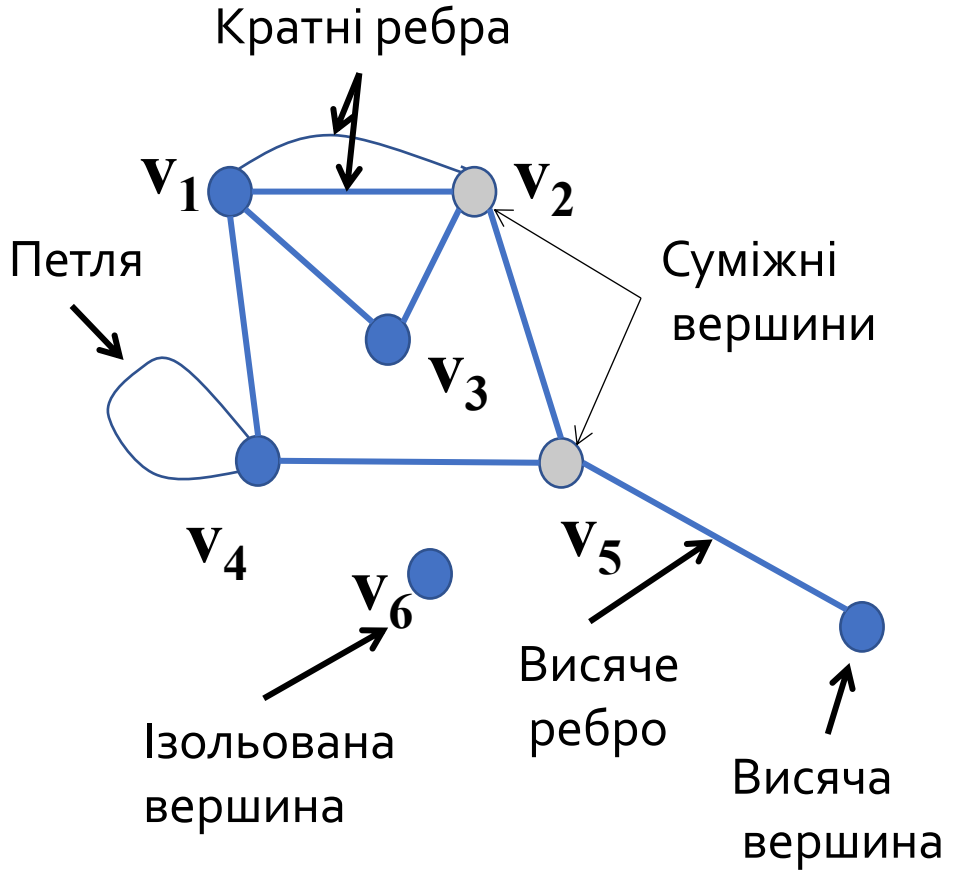
Ступінь вершини (Degree of a Vertex) - кількість ребер, інцидентних цій вершині

Інцидентність - це відношення між вершиною (v) і ребром (e)

Ребро e є інцидентним до вершини v (і навпаки), якщо вершина v є однією з його кінцевих точок

Шлях (Path) - послідовність ребер, що з'єднують дві вершини

Цикл (Cycle) - шлях, який починається і закінчується в одній і тій же вершині



Основна термінологія

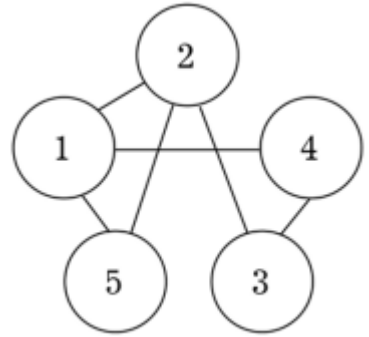
Неорієнтований граф (Undirected Graph) - ребра не мають напрямку, можна рухатися по ребрам в обидва боки

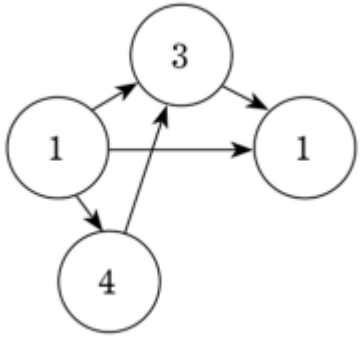
Орієнтований граф (Directed Graph, орграф) - це граф, ребра якого мають напрям.

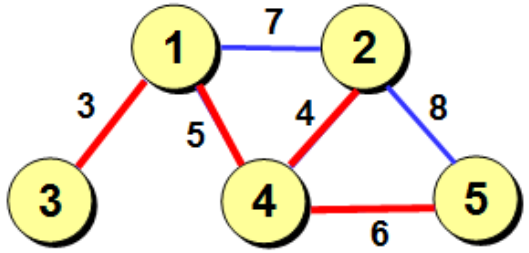
Ребра орграфа називаються **дугами**

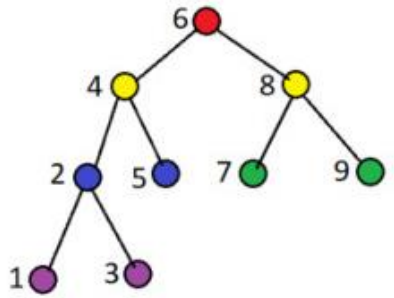
Зважений граф (Weighted Graph) - кожне ребро має асоційоване числове значення (вагу), що відображає відстань, вартість, час тощо

Дерево (Tree) - спеціальний випадок зв'язного, ациклічного (без циклів) неорієнтованого графа







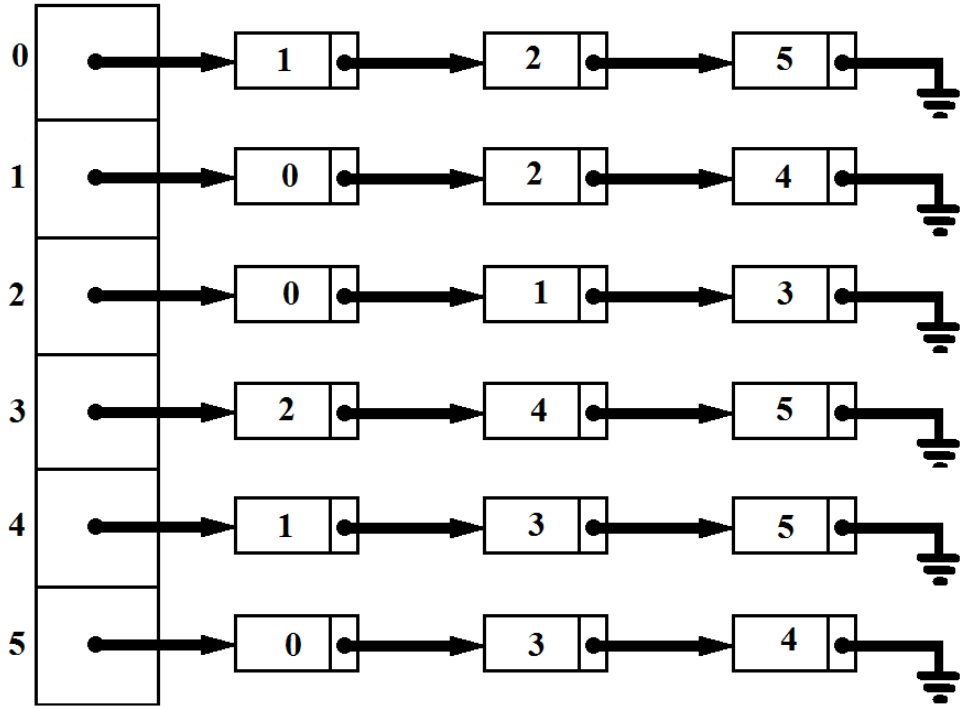


Способи представлення графа в пам'яті комп'ютера

- **Список суміжності (Adjacency List) -** найкращий для розріджених (sparse) графів (коли ребер мало)
- **Матриця суміжності (Adjacency Matrix) -** найкращий для щільних (dense) графів та швидкої перевірки зв'язку
- **Матриця інцидентності (Incidence Matrix) -** зручний для роботи з ребрами та в деяких теоретичних задачах

Список суміжності (Adjacency List)

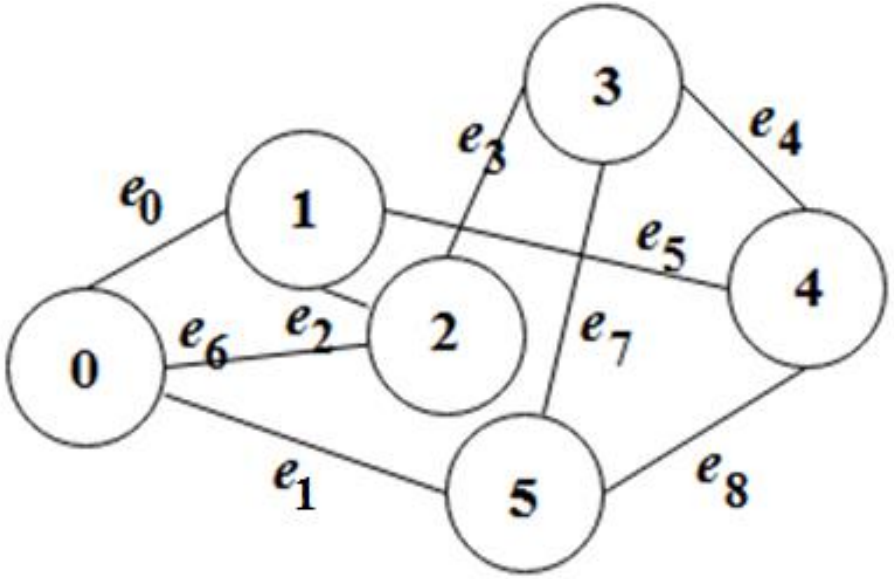
Список суміжності - представлення графу масивом зв’язного списку. Індекс масиву відповідає номеру вершини, а кожний елемент - це голова зв’язного списку із суміжних її вершин



Матриця суміжності (Adjacency List)

Матриця суміжності - це двовимірний масив (матриця) розміром V×V (вершин). Кожний рядок і стовпець представляють певну вершину:

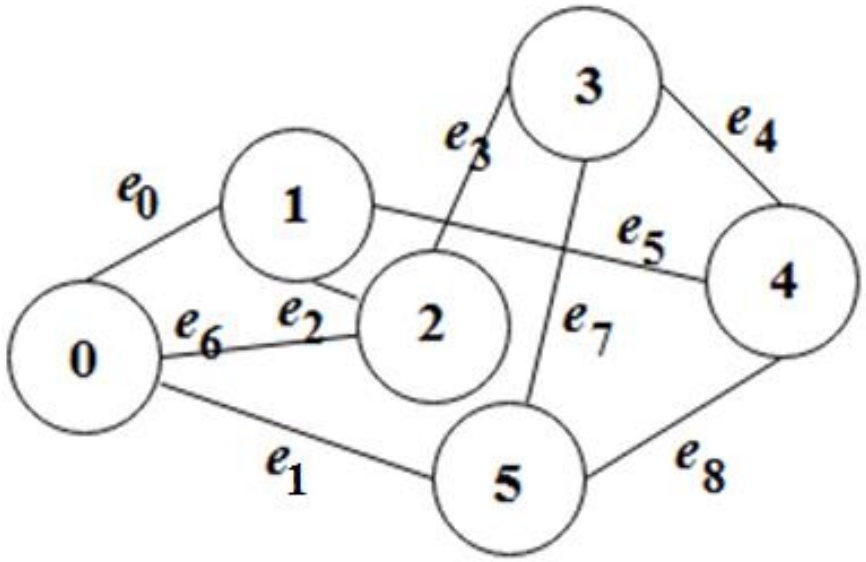
- A[i][j] = 1 (або вага ребра), якщо існує ребро від вершини *i* до вершини *j*
- A[i][j] = 0, якщо ребра немає



$$A = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

Матриця інцидентності (Incidence Matrix)

Матриця інцидентності – це двовимірний масив B розміром V×E. Кожний рядок відповідає певні вершині, а стовпець - ребру. Якщо значення будь-якого елемента B*ij* дорівнює 1, це означає, що ребро *j* інцидентне вершині *i*



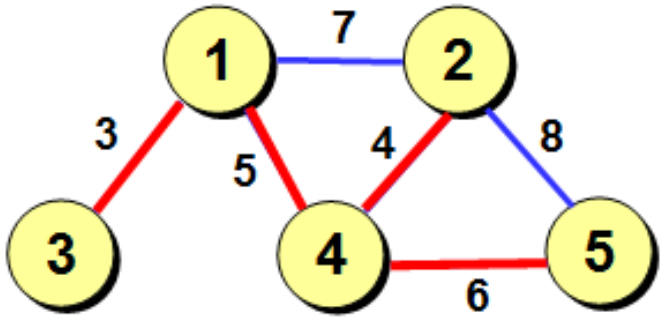
$$B = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

Матриця ваг

Матриця ваг - це двовимірний масив $W_{V \times V}$, в якому кожний елемент W_{ij} дорівнює вазі ребра (дузі), що з'єднує вершину *i* з вершиною *j* (виходить з вершини *i* та заходить у вершину *j*).

Даний граф є неспрямованим, тому його матриця ваг симетрична. $W[i, j]$ дорівнює вазі ребра (i, j), або ∞ (нескінченності), якщо ребро відсутнє.

Набір вершин: $V = \{1, 2, 3, 4, 5\}$.



	вершини	1	2	3	4	5
	1	0	7	3	5	∞
W=	2	7	0	∞	4	8
	3	3	∞	0	∞	∞
	4	5	4	∞	0	6
	5	∞	8	∞	6	0

Основні алгоритми на графах

Обхід графа

- **Пошук у ширину (Breadth-First Search, BFS) -** обходить граф по "шарах", знаходить найкоротші шляхи за кількістю ребер
- **Пошук у глибину (Depth-First Search, DFS) -** обходить граф максимально глибоко, перш ніж повернутися

Пошук найкоротшого шляху

- **Алгоритм Дейкстри (Dijkstra's Algorithm) -** знаходить найкоротший шлях від однієї вершини до всіх інших у графі з невід'ємними вагами ребер
- **Алгоритм Беллмана-Форда (Bellman-Ford Algorithm) -** знаходить найкоротший шлях у графі з від'ємними вагами ребер
- **Алгоритм Флойда-Уоршелла (Floyd-Warshall Algorithm) -** знаходить найкоротші шляхи між усіма парами вершин

Мінімальне остівне дерево (Minimum Spanning Tree, MST):

- **Алгоритм Прима (Prim's Algorithm) -** будує MST, починаючи з однієї вершини
- **Алгоритм Крускала (Kruskal's Algorithm) -** будує MST, обираючи ребра з найменшою вагою

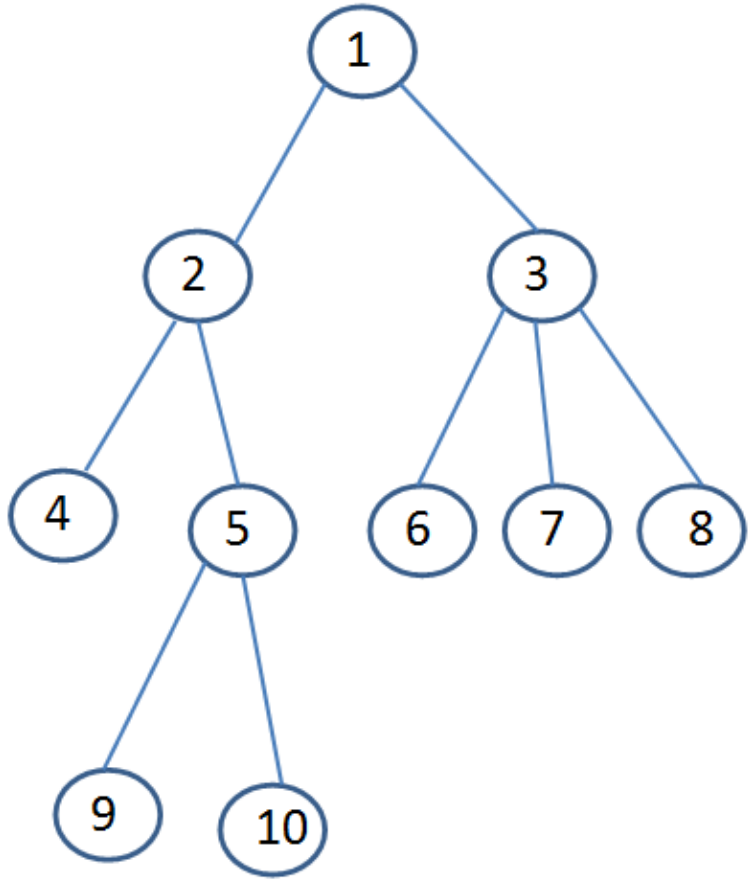
Дерево (Tree)

Дерево - це будь-який неорієнтований граф, який задовольняє двом критично важливим умовам

•**Зв'язність (Connected)** - між будь-якими двома вершинами (вузлами) графа існує **єдиний** шлях. Іншими словами, ви можете дістатися з будь-якої точки до будь-якої іншої

•**Ациклічність (Acyclic)** - граф не містить жодних циклів. Якщо ви починаєте рух з будь-якої вершини, ви ніколи не повернетеся до неї тим же шляхом, якщо не пройдете через одне й те саме ребро двічі

Кількість ребер - якщо дерево містить V вершин, воно завжди міститиме рівно V-1 ребро



Дерево. Термінологія

Хоча в математиці дерево може бути неорієнтованим (без кореня), у програмуванні та інформатиці найчастіше використовують **кореневі дерева (Rooted Trees)**

Корінь - найвища вершина, з якої починається дерево; усі інші вузли мають бути досяжні від кореня

Батько - Вершина, з якої виходить ребро до іншої вершини (дитини)

Дитина (нащадок) - вершина, до якої веде ребро від вершини-батька

Листок - вершина, яка не має жодних дітей (кінцевий вузол)

Піддерево - будь-яка частина дерева, яка сама по собі є деревом

Висота/Глибина - максимальна довжина шляху від кореня до найдальшого листка

Дерева використовуються всюди, де потрібно організувати ієрархію або забезпечити ефективний пошук даних:

- **Файлові системи (каталоги та підкаталоги)**
- **Дерева синтаксичного розбору (AST) (компілятори мов програмування)**
- **Двійкові дерева пошуку (BST) (для швидкого пошуку та сортування даних)**

Бінарне дерево (Binary Tree)

Бінарне дерево - це кореневе дерево, в якому кожна вершина (вузол) має **щонайбільше** дві "дитини" (нащадки), які зазвичай називаються **лівим** і **правим** нащадками.

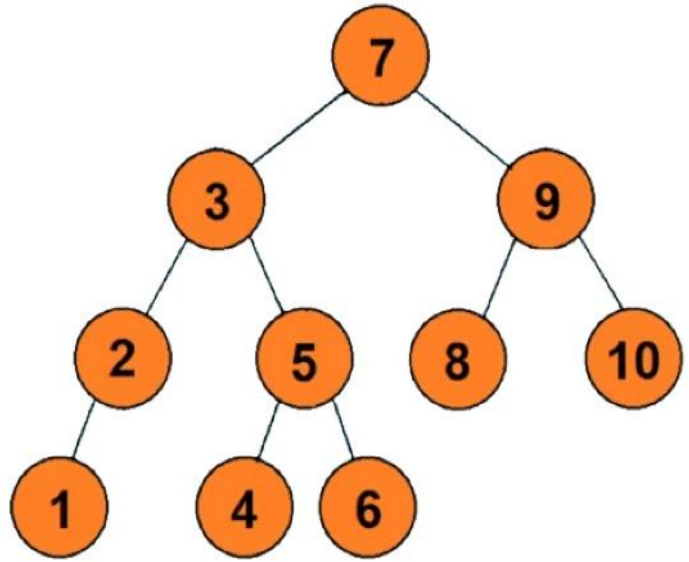
Бінарні дерева надзвичайно ефективні для організації та пошуку даних

Властивості:

•**Ліва та Права Дитина -** кожна вершина може мати 0, 1 або 2 нащадки.

•**Порядок важливий -** ліва дитина і права дитина **різні**. Навіть якщо вузол має лише одного нащадка, важливо, чи це ліва, чи права гілка.

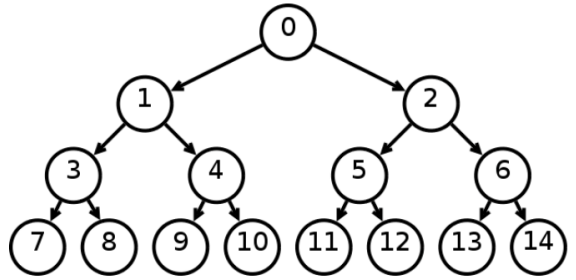
•**Ієрархія -** як і в будь-якому кореневому дереві, існує єдиний шлях від кореня до будь-якої іншої вершини.

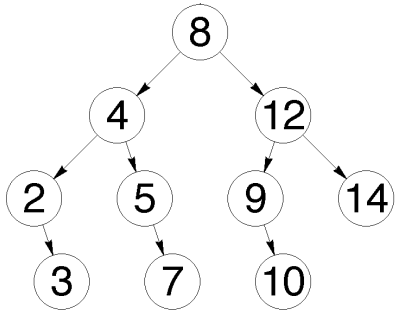


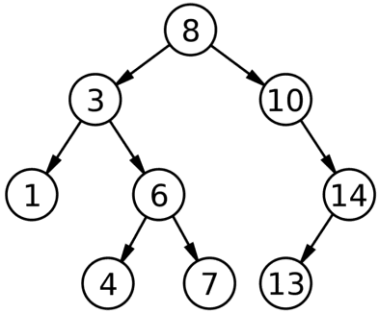
Типи бінарних дерев

Тип бінарного дерева визначає, як воно збалансоване або заповнене

Тип	Умова	Застосування
Повне (Full)	Кожна вершина має або **0**, або **2** нащадки	Використовується в деяких алгоритмах кодування
Ідеальне (Perfect)	Повне бінарне дерево, де всі листки розташовані на одному рівні (однаковій глибині)	Найкращий випадок для алгоритмів пошуку
Збалансоване (Balanced)	Різниця висот лівого та правого піддерев для будь-якого вузла не перевищує 1. (Наприклад, AVL-дерева, Червоно-чорні дерева)	Гарантує швидкий час доступу O(log N), де N - кількість вершин. Логарифмічна складність означає, що час роботи алгоритму зростає дуже повільно, оскільки щоразу, коли ви робите крок, ви відкидаєте половину елементів, що залишилися
Бінарне Дерево Пошуку (BST)	Значення **лівого** нащадка менше за значення батька, а значення **правого** нащадка більше за значення батька	Найбільш поширений тип для зберігання та пошуку впорядкованих даних



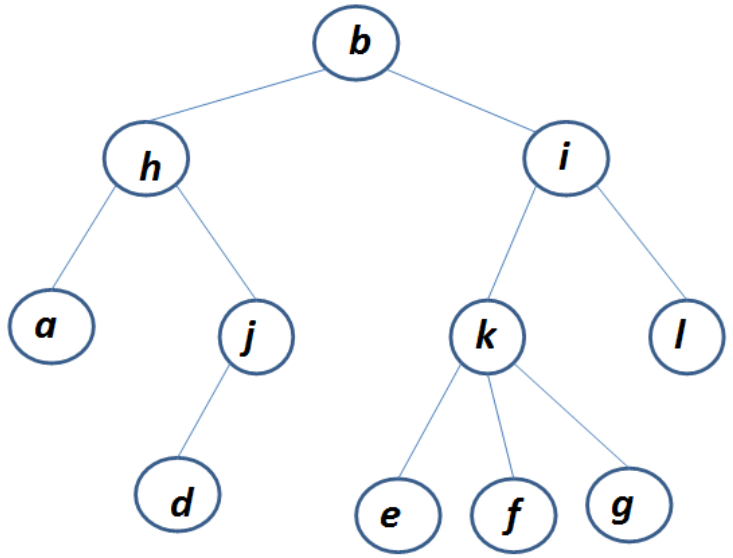


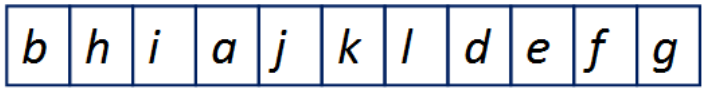


Пошукові алгоритми на графах та деревах

Обхід в ширину (Breadth-First Search, BFS)

Обхід дерева у ширину - це алгоритм, який використовується для проходження або пошуку у дереві (або графі), починаючи з кореня і досліджуючи **всі вузли на поточному рівні** перед тим, як перейти до вузлів наступного рівня





Принцип роботи алгоритму BFS

Для реалізації BFS використовується **черга (Queue)**

Етапи:

- **Ініціалізація -** створити чергу та додати до неї кореневий вузол
- **Обхід -** доки черга не порожня, повторювати:
  - витягнути (видалити) перший вузол із черги (**Dequeue**);
  - обробити цей вузол (наприклад, вивести його значення);
  - додати всіх нащадків цього вузла (спочатку лівого, потім правого) до кінця черги

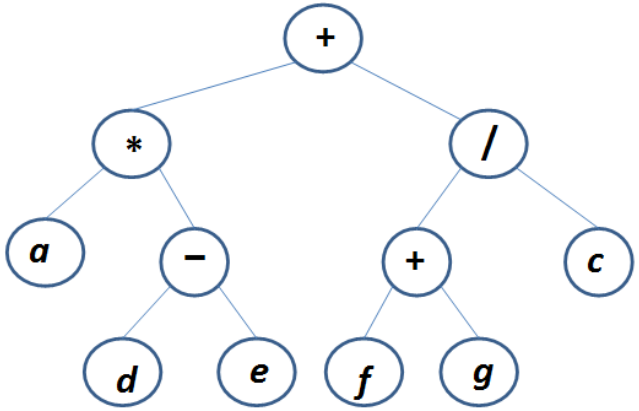
Приклад реалізації обходу дерева в ширину (BFS) за допомогою черги

```
def breadth_first_search(root):
        # Обробка пустого дерева
        if root is None:
        return []

      # Ініціалізація черги та результату
    queue = deque([root])
    result = []

      # Головний цикл обходу; повторюється, доки в черзі є вузли, які потрібно відвідати
    while queue:
        # Видалення вузла з початку черги (принцип FIFO: Першим прийшов - Першим вийшов)
        current_node = queue.popleft()

          # Додавання значення поточного вузла до результату обходу
        result.append(current_node.value)

          # Додавання нащадків поточного вузла в кінець черги

        # Додавання лівого нащадка (якщо існує)
        if current_node.left:
            queue.append(current_node.left)

        # Додавання правого нащадка (якщо існує)
        if current_node.right:
            queue.append(current_node.right)

    # Повернення результату
    return result
```

Обхід у глибину (Depth-First Search, DFS)

На відміну від BFS, який використовує чергу для обходу рівнями (пошарово), **DFS використовує стек** (або рекурсію) і рухається настільки **глибоко**, наскільки це можливо, вздовж однієї гілки, перш ніж "повернутися" назад (backtrack) і досліджувати наступну гілку

Дерево Виразів (Expression Tree) - це особливий тип бінарного дерева, який використовується для графічного представлення математичних або логічних виразів



Префіксний обхід:
+ * a – d e / + f g c

Постфіксний обхід:
a d e – * f g + c / +

Принцип роботи алгоритму DFS

Пріоритет глибини - алгоритм завжди прагне рухатися максимально далеко вздовж однієї гілки (шляху) у графі чи дереві

Використання стеку/рекурсії - DFS використовує структуру даних **стек** (LIFO, останній зайшов - перший вийшов) або, частіше, **рекурсію** для відстеження шляху та "запам'ятовування", куди повернутися

Етапи:

•Починаємо з вибраного вузла

•Відвідуємо вузол і позначаємо його як пройдений

•Вибираємо першого невідвіданого сусіда і негайно переходимо до нього, повторюючи процес (рекурсивний виклик)

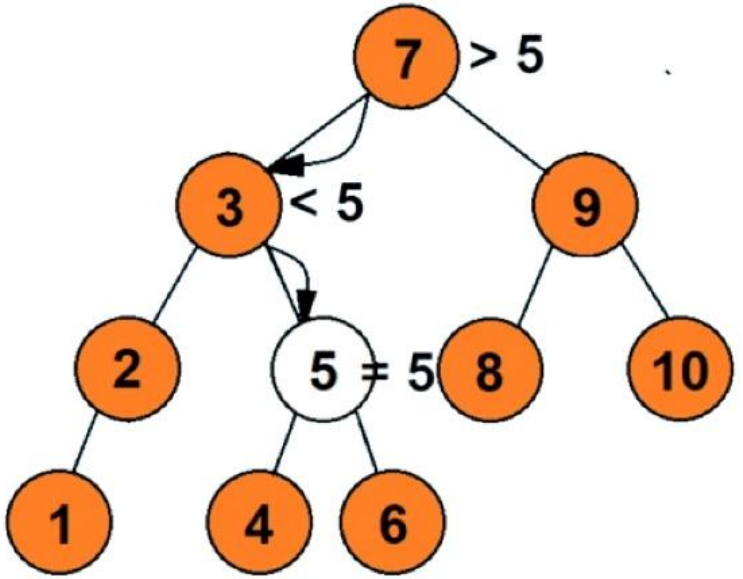
•Повернення (Backtracking) - коли алгоритм досягає вузла, який не має невідвіданих сусідів (тупик), він "повертається" назад (виходить з рекурсії або виштовхується зі стеку) до попереднього вузла, щоб дослідити його наступну невивчену гілку

•Завершення - процес триває, доки всі доступні вузли не будуть відвідані

Основні операції для бінарних дерев пошуку (BST)

Операція	Опис	Складність (середня)
Пошук (Search)	Перевірка, чи існує вузол із заданим значенням.	O(logn)
Вставка (Insertion)	Додавання нового вузла в потрібну позицію, зберігаючи властивість BST.	O(logn)
Видалення (Deletion)	Видалення вузла, зберігаючи цілісність дерева. Найскладніша операція, потребує заміни видаленого вузла його наступним елементом (in-order successor) або попереднім.	O(logn)
Знаходження мінімуму/максимуму	Найменше значення завжди в крайньому лівому вузлі, найбільше - у крайньому правому.	O(logn)

Пошук елемента



При пошуку елемента із заданим значенням починаємо з кореневого елемента:

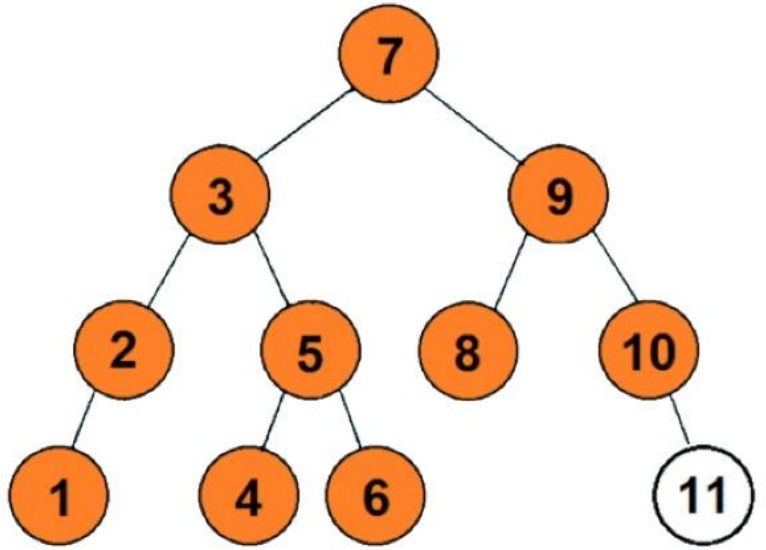
- якщо він дорівнює шуканому значенню, кореневий елемент і є шуканим, якщо ні - порівнюємо значення кореневого та шуканого;
- якщо потрібний елемент більший, переходимо до правого нащадка, якщо ні - до лівого;
- якщо елемент не знайдено, застосовуємо кроки 1 і 2, але вже до нащадка (правого або лівого) доти, доки елемент не буде знайдений

Наприклад, знайти елемент зі значенням 5:

- порівнюємо його з кореневим елементом; кореневий більше, тому ми переходимо до лівого нащадка, який має значення 3;
- порівнюємо шуканий елемент і елемент зі значенням 3; шуканий більше, тому потрібен правий нащадок елемента, що розглядається, а саме - елемент зі значенням 5;
- порівнюємо цього нащадка з шуканим; значення рівні, елемент знайдений.

Вставка елемента

Порівнюємо новий із кореневим (якщо його немає - новий елемент і є кореневим)
Якщо новий елемент:
• менше, переходимо до лівого нащадка, якщо його немає, новий елемент займає місце лівого нащадка, і алгоритм закінчено;
• більше чи дорівнює кореневому, переходимо до правого нащадка; якщо даного елемента немає, новий елемент займе місце правого елемента і алгоритм закінчено;
• для нового аналізованого елемента, який був правим або лівим з попереднього кроку, повторюємо кроки 1 і 2 до тих пір, поки елемент, що вставляється, не стане на своє місце



Наприклад необхідно вставити в дерево елемент зі значенням 11:
•порівнюємо з кореневим елементом 7; кореневий менше, тому переходимо до правого нащадка;
•наступний аналізований елемент має значення 9, що менше ніж новий 11; переходимо до правого нащадка;
•правий нащадок має значення 10, що менше 11; переходимо до першого елементу, а оскільки його немає, то новий елемент зі значенням 11 стає на це місце.