

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут телекомунікаційних систем

Кафедра телекомунікацій

До захисту допущено:

Завідувач кафедри

_____ Сергій КРАВЧУК

«__» _____ 2025 р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інженерія та програмування
інфокомунікацій»

спеціальності 172 «Телекомунікації та радіотехніка»

на тему: « Методи зберігання і оброблення даних в деревоподібних
структурах »

Виконала:

студентка IV курсу, групи ТЗ-12

Шишкова Олександра Юріївна _____

Керівник:

Ст. викладач кафедри ТК, к.ф.-м.н.

Руренко Олександр Григорович _____

Рецензент:

Доцент кафедри ІТТ НН ІТС, к.т.н., доцент

Правило Валерій Володимирович _____

Засвідчую, що у цій дипломній роботі
немає запозичень з праць інших авторів
без відповідних посилань.

Студентка _____

Київ – 2025 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Навчально-науковий інститут телекомунікаційних систем
Кафедра телекомунікацій

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 172 «Телекомунікації та радіотехніка»

Освітньо-професійна програма «Інженерія та програмування інфокомунікацій»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Сергій КРАВЧУК

«__» _____ 2025 р.

ЗАВДАННЯ
на дипломну роботу студенту
Шишків Олександрі Юрійові

1. Тема роботи «Методи зберігання і оброблення даних в деревоподібних структурах», керівник роботи Руренко Олександр Григорович, ст. викладач кафедри ТК, к.ф.-м.н., затверджені наказом по університету від «26» травня 2025 р. № 1755-с.
2. Термін подання студентом роботи 9 червня 2025 р.
3. Вихідні дані до роботи: програмне забезпечення: середовище розробки Visual Studio Code, мова програмування Python, бібліотека машинного навчання Scikit-learn; методичні матеріали: документація до бібліотеки Scikit-learn, наукові джерела з теорії деревоподібних структур даних, приклади реалізації алгоритмів дерева рішень.
4. Зміст роботи: ознайомитися з основними типами деревоподібних структур даних та їх застосуванням у сфері телекомунікацій; проаналізувати принципи роботи бінарного дерева пошуку і дерева рішень; вивчити функціональні можливості бібліотеки Scikit-learn для побудови моделей на основі дерева рішень; реалізувати класифікаційну модель дерева рішень для оцінки якості телекомунікаційного сигналу; провести візуалізацію та інтерпретацію побудованої моделі на основі вхідних параметрів сигнал

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо):

1. Вступ
2. Мета та завдання дипломної роботи
3. Актуальність теми
4. Об'єкт та предмет дослідження
5. Теоретичні основи деревоподібних структур даних
6. Порівняльна характеристика бінарного дерева пошуку та дерева рішень
7. Демонстрація практичної реалізації моделі дерева рішень у Python
8. Візуалізація дерева рішень (графічне зображення побудованої моделі)
9. Висновки до дипломної роботи

6. Дата видачі завдання 20.02.2025

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Робота над першим розділом	20.02.2025	Виконано
2	Робота над другим розділом	10.04.2025	Виконано
3	Робота над практичною частиною (розділ 3)	14.04.2025	Виконано
4	Продовження практичної частини (розділ 4)	18.05.2025	Виконано
5	Оформлення дипломної роботи	06.06.2025	Виконано

Студент

Олександра ШИШКОВА

Керівник

Олександр РУРЕНКО

РЕФЕРАТ

Пояснювальна записка до дипломної роботи містить 72 сторінки, 19 ілюстрацій, 2 таблиці, 1 додаток, 35 бібліографічне найменування.

Метою роботи є дослідження деревоподібних структур даних та реалізація моделі дерева рішень за допомогою мови програмування Python. Використання бібліотеки Scikit-learn для задач класифікації в сфері телекомунікацій.

У роботі розглянуто основні види дерев: бінарні дерева, бінарні дерева пошуку та дерева рішень. Проведено теоретичний аналіз, огляд принципів побудови, а також практичне моделювання дерев у Python. Було побудовано та візуалізовано модель дерева рішень для класифікації якості телекомунікаційного сигналу на основі параметрів: сила сигналу, рівень шуму та втрата пакетів.

Отримані результати показали можливість ефективного застосування деревоподібних структур у задачах телекомунікаційного аналізу, що відкриває перспективи для їх впровадження в системах моніторингу якості зв'язку.

Результати рекомендовано використовувати в навчальному процесі та при розробці програмного забезпечення для телекомунікаційного аналізу даних.

Ключові слова: дерево рішень, бінарне дерево пошуку, Python, Scikit-learn, телекомунікації, алгоритми класифікації, моделювання, машинне навчання, структура даних, сигнал, шум, втрата пакетів, якість зв'язку, візуалізація, аналіз.

ABSTRACT

The explanatory note to the thesis comprises 72 pages, 19 illustrations, 2 tables, 1 appendix, and 35 bibliographic entries.

The objective of this work is to investigate tree-like data structures and implement a decision tree model using the Python programming language. It involves the application of the Scikit-learn library for classification tasks in the telecommunications sector.

The thesis examines the main types of trees: binary trees, binary search trees, and decision trees. It conducted a theoretical analysis, an overview of construction principles, and practical tree modeling in Python. A decision tree model was built and visualized to classify telecommunication signal quality based on parameters such as signal strength, noise level, and packet loss.

The obtained results demonstrate the possibility of effectively applying tree-like structures in telecommunications analysis tasks, which opens up prospects for their implementation in communication quality monitoring systems.

The results are recommended for use in the educational process and in the development of software for telecommunications data analysis.

Keywords: decision tree, binary search tree, Python, Scikit-learn, telecommunications, classification algorithms, modeling, machine learning, data structure, signal, noise, packet loss, communication quality, visualization, analysis.

ЗМІСТ

ВСТУП	9
РОЗДІЛ 1	10
ЗАГАЛЬНІ ВІДОМОСТІ ПРО СТРУКТУРИ ДАНИХ ТА АЛГОРИТМИ.....	10
1.1 Історія та походження структур даних та алгоритмів	10
1.2 Поняття про структури даних	11
1.3 Класифікація структур даних.....	13
1.3.1 Лінійні структури даних.....	14
1.3.2 Нелінійні структури даних.....	18
1.3.3 Відмінності між лінійними та нелінійними структурами даних	23
1.4 Алгоритми в структурах даних.....	24
Висновки	26
РОЗДІЛ 2	28
ДЕРЕВОПОДІБНІ СТРУКТУРИ ДАНИХ.....	28
2.1 Загальні поняття про деревоподібні структури даних	28
2.1.1 Властивості деревоподібних структур даних	32
2.2 Бінарні структури даних.....	33
2.2.1 Властивості бінарного дерева:.....	36
2.2.2 Операції над бінарним деревом.....	38
2.3 Бінарні дерева пошуку.....	40
2.3.1 Алгоритми обробки вузлів у BST.....	41
2.3.2 Застосування BST	51
2.3.3 Аналіз переваг та недоліків бінарних дерев пошуку	53
2.4 Древа рішень.....	54
2.4.1 Типи дерев рішень	56
2.4.2 Алгоритми побудови дерева рішень	57
2.4.3 Оптимізація дерев рішень: техніки обрізання дерева	59
2.4.4 Характерні особливості дерев рішень: переваги та обмеження.....	60

2.4.5 DecisionTreeClassifier	62
Висновки	63
РОЗДІЛ 3	65
ВИКОРИСТАННЯ БІБЛІОТЕКИ SCIKIT-LEARN ДЛЯ ПОБУДОВИ ДЕРЕВ РІШЕНЬ	65
3.1 Загальні характеристики бібліотеки Scikit-learn	65
3.2 Інструменти Scikit-learn для побудови дерева рішень	66
3.2.1 Порівняльна таблиця: Decision Tree, Logistic Regression, Random Forest	67
3.3 Практичне застосування Scikit-learn у телекомунікаціях	68
Висновки	68
РОЗДІЛ 4	70
ПОБУДОВА МОДЕЛІ ДЕРЕВА РІШЕНЬ ДЛЯ КЛАСИФІКАЦІЇ ЯКОСТІ ЗВ'ЯЗКУ	70
Висновки	72
ЗАГАЛЬНІ ВИСНОВКИ ПО РОБОТІ	74
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	76
ДОДАТОК 1	80
ДОДАТОК 2	81

ПЕРЕЛІК СКОРОЧЕНЬ

BT	Binary Tree
BST	Binary Search Tree
CART	Classification and Regression Tree
DS	Data Structure
DSA	Data Structures and Algorithms
DT	Decision Tree
ID3	Iterative Dichotomiser 3
MSE	Mean Squared Error
TDS	Tree Data Structure

ВСТУП

У сучасному світі зростає потреба у швидкій та надійній обробці великих обсягів даних. Через швидкий розвиток телекомунікаційних систем, штучного інтелекту та автоматизованих рішень, дедалі важливішим стає використання ефективних структур зберігання та оброблення даних. Одним із передових напрямів у цій сфері є застосування деревоподібних структур даних, які забезпечують гнучке й оптимізоване представлення взаємозв'язаних елементів даних.

Аналіз сучасної технічної літератури та практики провідних ІТ-компаній (зокрема Google, Meta, Amazon) свідчить про активне використання дерев рішень, бінарних дерев пошуку та їх модифікацій у задачах класифікації, пошуку, маршрутизації, а також у машинному навчанні. Наприклад, у телекомунікаційних системах дерева використовуються для побудови таблиць маршрутизації, у базах даних – для організації індексації, а в системах підтримки прийняття рішень – для моделювання логіки поведінки.

Ця дипломна робота присвячена аналізу, реалізації та порівнянню деревоподібних структур. Було розглянуто бінарні дерева, бінарні дерева пошуку та дерева рішень, з використанням мови програмування Python та бібліотеки для машинного навчання Scikit-learn. Новизна цього дослідження полягає у порівнянні роботи класичних структур даних із алгоритмами сучасного машинного навчання, які були розглянуті на прикладах телекомунікаційних задач.

Отримані результати можуть стати корисними для автоматизованих систем обробки трафіку, розумних систем управління мережею, а також для програм, що займаються аналізом даних у сфері інформаційної безпеки.

РОЗДІЛ 1

ЗАГАЛЬНІ ВІДОМОСТІ ПРО СТРУКТУРИ ДАНИХ ТА АЛГОРИТМИ

1.1 Історія та походження структур даних та алгоритмів

Структури даних та алгоритми (англ. Data Structures and Algorithms, DSA) є важливою частиною в галузі комп'ютерних наук. Це формує фундамент для розуміння того, як ефективно організовувати, зберігати та обробляти інформацію. Структури даних допомагають будувати масштабовані, швидкі та надійні програмні системи. Вони лежать в основі майже всіх обчислювальних завдань - від простих програм до складних систем зі штучним інтелектом чи аналізом великих обсягів даних.

Через зростання обсягів даних та вимог до продуктивності програмного забезпечення, вибір правильної структури даних та алгоритму є надзвичайно важливим. Від вибору правильної структури даних і алгоритму залежить те, як ефективно будуть використовуватися пам'ять, процесорний час і пропускна здатність мережі. Правильно підібрані рішення допомагають значно прискорити обробку даних і знизити витрати, тоді як помилки на даному етапі можуть уповільнити роботу системи й призвести до марного витрачання ресурсів.

Структури даних, як ідея впорядкованого зберігання та доступу до інформації, набула формулювання разом з появою перших програмованих комп'ютерів у середині XX століття. У 1940-х роках в роботах Алана Тюрінга та Джона фон Неймана були закладені основи обчислювальної моделі, яка потребувала впорядкованого подання даних для їх ефективної обробки.

З розвитком мов програмування у 1950–60-х роках (наприклад, Fortran, Lisp, Algol) постала потреба в ефективних способах організації даних у пам'яті комп'ютера. Саме тоді з'явилися перші базові структури даних: масиви, зв'язані списки, стеки та черги. Вони були необхідними для вирішення

типових задач, таких як обробка рядків, керування пам'яттю чи реалізація рекурсивних процесів.

Одним із важливих етапів стала праця Дональда Кнута «Мистецтво програмування» (*The Art of Computer Programming*, перше видання - 1968 рік). У ній вперше було системно викладено методи побудови алгоритмів і відповідних структур даних, а також було започатковано традицію математичного аналізу і їх ефективності, що стало основою сучасного поняття обчислювальної складності.

У 1970–80-х роках розвиток програмного забезпечення, баз даних та компіляторів призвів до появи складніших структур, таких як дерева (бінарні дерева пошуку, В-дерева), хеш-таблиці, графи тощо. Ключову роль відіграли деревоподібні структури, які дозволили ефективно вирішувати задачі пошуку, сортування, ієрархічного зберігання даних та побудови логічних рішень. [1, 2]

1.2 Поняття про структури даних

Структура даних (англ. Data Structures, DS) - це спосіб для організації, обробки, пошуку та зберігання даних у комп'ютерній системі. Вона описує, як елементи даних пов'язані між собою і які операції можна над ними виконувати для ефективного управління ними. Структури даних (DS) мають важливе значення для комп'ютерних наук і є основою створення програмних систем та алгоритмів. Існують базові та розширені типи DS, кожен з яких призначений для впорядкування даних відповідно до визначеної мети. [26]

З технічної точки зору, структура даних (DS) реалізується за допомогою змінних, покажчиків, масивів, об'єктів або інших механізмів, які надає мова програмування. У узагальненому сенсі DS визначає не лише спосіб зберігання, а й набір допустимих операцій - наприклад, вставка нового елемента, видалення, пошук, сортування тощо.

Важливим також те, що DS допомагають ефективно використовувати апаратні ресурси - пам'ять, процесорний час, мережеві ресурси тощо. У

великих системах, таких як вебсервіси, бази даних чи системи зі штучним інтелектом, правильний вибір структури даних безпосередньо впливає на те, наскільки добре система масштабуватиметься й наскільки швидко працюватиме.

Загалом, поняття DS охоплює не тільки спосіб представлення інформації, а й логіку її обробки, що робить цю концепцію головною в будь-якому програмному або інженерному рішенні. [28, 30]

Ключові особливості DS:

- Зберігання даних: DS забезпечують організоване зберігання даних, що полегшує управління даними та ефективний доступ до них.
- Організація даних: вони організовують дані у структурованому форматі, наприклад, лінійному (масиви, зв'язані списки) або нелінійному (дерева, графіки), що полегшує маніпулювання даними.
- Обробка даних: DS дозволяють ефективно та швидко виконувати такі операції, як додавання, видалення або обхід елементів. Кожна така структура оптимізована для певних завдань, що підвищує продуктивність.
- Управління пам'яттю: DS ефективно керують використанням пам'яті, а зв'язані списки та дерева, динамічно розподіляють пам'ять за потребою.
- Однорідні дані: багато DS зберігають елементи даних одного типу, що робить дані більш передбачуваними та зручними для обробки.
- Індексування: масиви дозволяють ефективно та швидко знаходити елемент за його позицією, що є зручним, коли потрібно швидко істатись до потрібної інформації.
- Послідовний доступ: лінійні DS, такі як масиви або зв'язані списки, забезпечують послідовний доступ до елементів, полегшуючи впорядковану обробку даних.

- Динамічний розмір: зв'язані списки та динамічні масиви можуть динамічно змінювати свій розмір, щоб пристосуватись до різних обсягів даних.
- Ієрархічне представлення: дерева та графи відображають ієрархічні зв'язки між елементами, що корисно для моделювання складних систем.
- Підтримка різноманітних операцій: DS підтримують різноманітні операції, включаючи пошук, сортування, злиття та розділення, що робить їх універсальними інструментами для маніпулювання даними.
- Простота реалізації: багато DS підтримуються стандартними бібліотеками в мовах програмування, забезпечуючи прості у використанні реалізації для поширених структур.[26]

1.3 Класифікація структур даних

Класифікація структур даних (DS) допомагає краще зрозуміти для чого вони потрібні, як вони влаштовані та як можна ефективно використовувати їх в різних обчислювальних завданнях. Залежно від способу розміщення елементів у пам'яті та характеру зв'язків між ними, DS поділяють на лінійні й нелінійні, а також на статичні й динамічні. Така класифікація дозволяє систематизувати знання про базові типи DS і полегшує вибір відповідної моделі для розв'язання конкретної задачі. На рисунку нижче представлено загальну схему класифікації структур даних (DS), яка відображає їх поділ за лінійністю та способом організації: [2]

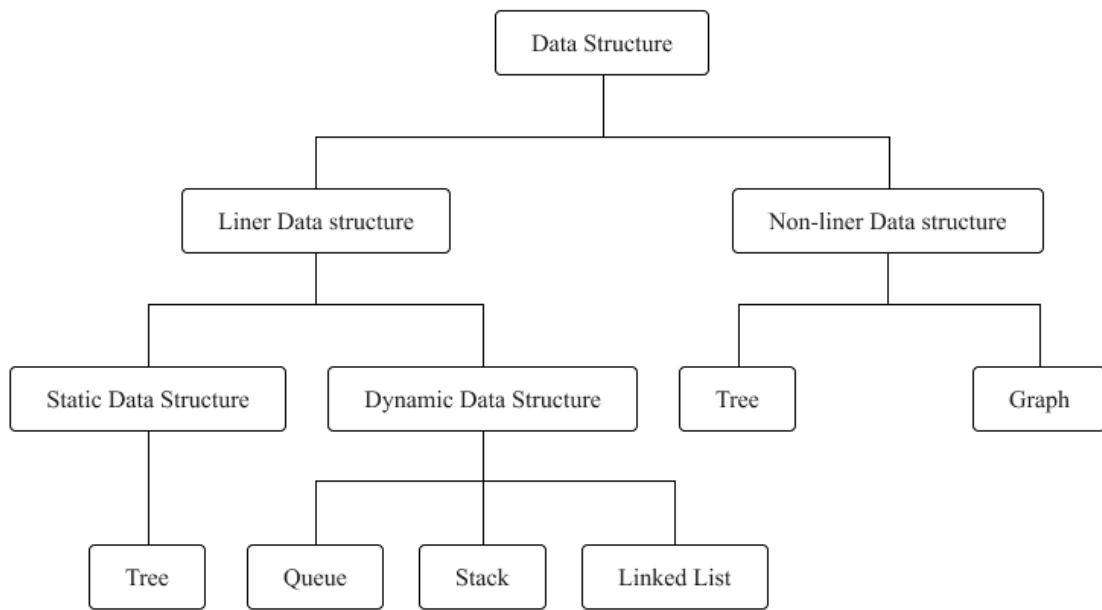


Рис. 1.1 Класифікація структур даних

1.3.1 Лінійні структури даних

Лінійна структура даних (англ. Linear Data Structure, лінійна DS) - це тип структури даних (DS), в якій елементи розташовані послідовно, один за одним. Кожен елемент пов'язаний з попереднім і наступним елементом, що утворює лінійну послідовність. Завдяки такому розташуванню можна легко виконувати операції обходу, вставки та видалення.

Лінійні DS є вагомими, тому що вони забезпечують простий та ефективний спосіб зберігання та доступу до даних. Вони використовуються в різних алгоритмах і додатках, від зберігання даних до складних методів розв'язання задач. Прикладами таких DS є масиви, зв'язані списки, стеки та черги. Кожна з цих структур має унікальні властивості та варіанти використання.

Загалом, лінійні DS допомагають впорядковувати дані у послідовний спосіб, що полегшує перегляд та керування ними. Розуміння лінійних DS є важливим для формування більш складних структур та алгоритмів. [4]

Ключові характеристики лінійних структур даних:

- Послідовне розташування: у лінійних DS елементи розташовані в лінійній послідовності, кожен з елементів пов'язаний з попереднім і наступним елементом.
- Однорівневе зберігання даних: на відміну від ієрархічних структур, таких як дерева, елементи даних зберігаються на одному рівні.
- Простий та ефективний перехід: перехід між елементами зазвичай вимагає одного циклу або ітерації.
- Ефективне використання пам'яті: лінійні DS економно використовують пам'ять для зберігання, часто вимагаючи суміжних ділянок пам'яті.
- Операції вставки та видалення: ці операції можуть виконуватися відносно ефективно, особливо у структурах даних, таких як зв'язані списки, завдяки їхній динамічній природі та гнучкості в управлінні пам'яттю.
- Фіксований розмір або динамічність: деякі лінійні DS, а саме масиви, мають фіксований розмір, тоді як інші, такі як зв'язані списки, можуть динамічно зменшуватися або збільшуватися.

Ці характеристики роблять лінійні DS базовими для різних додатків, забезпечуючи прості, але потужні засоби для управління та маніпулювання даними.

Лінійні DS існують у різних формах, кожна має свої унікальні характеристики та варіанти застосування. Розглянемо нижче деякі з найпоширеніших типів:

- Масив (англ. Array) - це лінійна структура даних (лінійна DS), яка зберігає набір елементів, частіше одного типу даних, у суміжних комірках пам'яті. До кожного елементу масиву можна отримати доступ за допомогою індексу, що в свою чергу полегшує пошук та маніпулювання даними.

- Зв'язані списки (англ. Linked Lists) - це лінійна структура даних (лінійна DS), яка складається з вузлів, де кожен вузол містить елемент даних і посилання (лінк) на наступний вузол у послідовності. Зв'язані списки, на відміну від масивів, не зберігають елементи у суміжних комірках пам'яті. Замість цього, кожен елемент вказує на наступний, утворюючи ланцюгову структуру з'єднаних комірок пам'яті. Завдяки такій динамічній будові зі списками можна легко й ефективно додавати або видаляти елементи. Вони особливо корисні в ситуаціях, коли заздалегідь невідомо, скільки буде елементів, або коли часто доводиться щось вставляти чи видаляти.
- Стек (англ. Stack) - це лінійна структура даних (лінійна DS), яка працює за принципом «останній прийшов, перший пішов» або Last In, First Out (LIFO). У даному випадку елементи стеку розташовані один за одним. Це означає, що останній елемент, який був доданий до стеку, є першим, який буде видалено. Стеки використовуються в управлінні викликами функцій у програмуванні, обчисленні виразів та синтаксичних розборах. Вони можуть реалізовуватися за допомогою масивів або зв'язаних списків, що забезпечує простий спосіб керування даними.
- Черги (англ. Queue) - це лінійна структура даних (лінійна DS), яка працює за принципом «першим прийшов, першим пішов» або First In, First Out (FIFO). В чергах перший елемент, який був доданий до черги, першим видаляється. Черги широко використовуються в плануваннях завдань, буферизації та в алгоритмах пошуку в ширину. Вони також можуть динамічно збільшуватися і зменшуватися при додаванні або видаленні елементів. [3]

Найпоширеніші операції, що виконуються в лінійних DS:

Лінійні DS, такі як масиви, зв'язані списки, стеки та черги, є базовими елементами в комп'ютерних науках. Вони підтримують різні, які допомагають

працювати з даними та змінювати їх. Ці операції необхідні для ефективного управління даними і є основою для більш складних алгоритмів.

Однією з найпоширеніших операцій є обхід (англ. Traversal), який передбачає систематичний доступ і послідовну обробку кожного елемента DS. Це є головним для завдань пошуку певного елемента, оновлення вже наявних даних або відображення всіх елементів.

Існує два основні типи операцій, за допомогою яких можна змінювати DS - це вставка (англ. Insertion) і видалення (англ. Deletion). Вставка означає додавання нового елемента у визначену позицію, що часто вимагає зміщення існуючих елементів, щоб розмістити новий елемент. Видалення передбачає вилучення елемента з певної позиції, що вимагає переміщення наступних елементів, щоб закрити прогалину, яка утворилася. Для масивів ці операції використовують оператор індексування.

Окрім базових маніпуляцій, лінійні DS також підтримують операції впорядкування та запиту даних. Пошук (англ. Searching) має знайти позицію певного елемента у структурі, використовуючи такі алгоритми, як лінійний пошук для невідсортованих даних або бінарний пошук для відсортованих даних. Сортування (англ. Sorting) впорядковує елементи в певному порядку, за зростанням або спаданням, використовуючи різні алгоритми, такі як бульбашкове сортування, швидке сортування, сортування злиттям або сортування вставкою, кожен з яких має свої певні характеристики.

Також злиття (англ. Merging) об'єднує дві структури даних в одну, при цьому зберігає порядок елементів. Такий підхід часто використовується в парі з сортуванням, щоб ефективно об'єднувати списки або масиви. Реверсування (англ. Reversing) інвертує порядок елементів у структурі даних і робить останній елемент першим і навпаки.

Ці операції забезпечують основу для ефективної роботи з даними в лінійних структурах (лінійних DS), що лежать в основі широкого спектру обчислювальних задач і методів обробки даних. [2, 3]

1.3.2 Нелінійні структури даних

Нелінійна структура даних (нелінійні DS) - це спосіб організації даних, у якій елементи не розташовуються один за одним в лінійному порядку, як у масивах, стеках або чергах. Замість цього вони зберігаються у вигляді складних зв'язків, що можуть відображатися як ієрархії або багатовимірні мережі. Такий підхід дозволяє ефективно працювати з даними, які мають складну структуру.

На відміну від лінійних DS, де кожен елемент має чітко визначене місце у послідовності, нелінійні структури передбачають наявність одного або кількох зв'язків між елементами, що дозволяє ефективно реалізовувати такі реальні концепції, як дерева (наприклад, файлові системи або ієрархії в базах даних) та графи (соціальні мережі, маршрутизація в мережах, моделювання транспортних систем тощо). [5]

Основними характеристиками нелінійних DS :

Нелінійні DS мають свої особливі властивості, завдяки яким вони можуть ефективно показувати складні зв'язки між елементами. Головною рисою нелінійних DS є те, що елементи розташовані не послідовно. На відміну від лінійних DS, елементи не зберігаються в безперервних ділянках пам'яті. Натомість логічні зв'язки між ними реалізуються через вказівники або посилання. Хоча це й робить їх структуру складнішою, це забезпечує більшу гнучкість у доступі та маніпулюванні даними, що дозволяє елементам знаходитися в різних місцях пам'яті.

Ще одна важлива риса нелінійних структур даних - це складна мережа зв'язків між елементами. Один елемент може бути пов'язаний одразу з кількома іншими, що дозволяє створювати розгалужені чи навіть циклічні структури. Завдяки цьому можна моделювати складні взаємозв'язки і відносини між об'єктами, які неможливо передати за допомогою лінійних структур.

Завдяки складним зв'язкам у нелінійних структурах даних, їх можна обходити по-різному. Оскільки відносини між елементами не є лінійними, існують варіанти проходження структури. Це може бути обхід в глибину (DFS), обхід в ширину (BFS), а також пошук за пріоритетами, що дозволяє адаптувати алгоритми для вирішення специфічних задач пошуку, оптимізації або аналізу залежностей.

З комплексною топологією та гнучкістю обходу пов'язана і відсутність уніфікованого порядку. У нелінійних DS немає чітко визначених "наступного" або "попереднього" елементів, як це є у послідовних списках. Порядок доступу до даних повністю задається структурною логікою (наприклад, батько-дитина в дереві) або конкретним алгоритмом обходу, який використовується для розв'язання певної задачі. [6]

Нелінійні структури особливо добре підходять для певних завдань, бо можуть моделювати складні зв'язки між даними. Вони ефективні, коли потрібно працювати з ієрархіями, мережами або асоціаціями. Наприклад, дерева рішень часто використовують у штучному інтелекті для класифікації та прогнозування, а графи - незамінні для задач, як маршрутизація, аналіз соціальних мереж чи моделювання складних систем.

Графова структура даних:

Графова структура даних (англ. Graph Data Structure, DS) - це нелінійна структура даних, що складається з вершин і ребер. Вершини іноді також називають вузлами, а ребрами - лініями або дугами, які з'єднують будь-які дві вершини графа. Більш формально граф складається з множини вершин(V) та множини ребер(E). Граф позначається $G(V, E)$.

Граф, як нелінійна структура даних (DS), складається з двох базових компонентів, що визначають його будову та дозволяють моделювати різноманітні взаємозв'язки: вершин та ребер.

Вершини (англ. Vertices): вершини є основними блоками будь-якого графа. Їх також називають вузлами (nodes). Кожна вершина представляє окремий об'єкт або сутність у системі, яку моделює граф. Вершини можуть

мати або не мати певне позначення (мітку), що дозволяє ідентифікувати їх або зберігати додаткову інформацію, пов'язану з цим об'єктом.

Ребра (англ. Edges): ребра в графі - це лінії, які з'єднують вершини і показують, що між ними існує якийсь зв'язок. Вони можуть поєднувати будь-які дві вершини в будь-який спосіб, без жорстких обмежень щодо кількості чи розташування. Іноді ребра називають дугами, особливо коли йдеться про орієнтовані графи — там ребро має напрямок від однієї вершини до іншої. Як і вершини, ребра можуть містити додаткову інформацію, наприклад, про тип чи вагу зв'язку.

Однією з ключових класифікацій графів є поділ за спрямованістю їхніх ребер, що визначає характер зв'язку між вершинами та суттєво впливає на сфери застосування графа. Неорієнтований граф характеризується ребрами, які не мають конкретного напрямку: якщо ребро з'єднує вершини А і В, це означає двосторонній зв'язок, що дозволяє переміщатися як з А до В, так і з В до А. Такі графи ідеально підходять для моделювання симетричних відносин, як-от дружні зв'язки в соціальних мережах або дорожні мережі, де дороги є двосторонніми. На противагу цьому, у спрямованому графі (або диграфі) кожне ребро має чітко визначений напрямок, який зазвичай позначається стрілкою; наприклад, ребро від А до В означає, що перехід можливий лише в цьому напрямку, але не обов'язково навпаки. [6]

Графи також розрізняють за наявністю додаткової інформації на їхніх ребрах. У зваженому графі кожному ребру присвоюється числове значення, або «вага». Це значення, як правило, відображає певний вимірюваний атрибут, пов'язаний з переходом по цьому ребру, наприклад, вартість, відстань, час або пропускну здатність. Наприклад, на дорожній карті ребра можуть позначати відстань між містами або час у дорозі. Для таких випадків існують спеціальні алгоритми - як Дейкстри чи Беллмана-Форда - які допомагають знайти найкращий шлях, враховуючи ці «ваги». А от у незважених графах ребра не мають ваги - всі переходи вважаються однаковими за «вартістю». Такі графи

простіші для аналізу і використовуються, коли важливо лише, чи є зв'язок між вершинами, а не наскільки він «важливий» чи «довгий».

Додаткові розрізнення між графами базуються на наявності певних структурних патернів. Циклічний граф містить щонайменше один цикл - шлях, який починається і завершується в одній і тій же вершині, проходячи через інші проміжні вершини. Такий тип графа часто зустрічається у мережевій маршрутизації або у ланцюгах залежностей, які можуть призводити до рекурсивних викликів. Натомість ациклічний граф повністю позбавлений циклів. Особливо важливим є спрямований ациклічний граф (DAG), у якому всі ребра мають напрямок і відсутні будь-які цикли. DAG широко застосовуються для моделювання процесів із залежностями, що вимагають виконання у чітко визначеному порядку без утворення циклічних залежностей - наприклад, у плануванні завдань, оптимізації компіляторів чи системах контролю версій. Іншою структурною категорією є повний граф - граф, у якому кожна пара вершин з'єднана унікальним ребром. Такі графи є щільними і відображають ситуації, коли кожна сутність має прямий зв'язок з усіма іншими. [30]

Нарешті, графи класифікуються за їхньою зв'язністю. Зв'язний граф - це такий граф, у якому існує шлях між будь-якою парою вершин, що означає його утворення єдиного, цілісного компонента. Якщо ж граф не є зв'язним, він називається незв'язним і складається з двох або більше окремих «зв'язних компонентів», між якими немає ребер. Ця різниця має велике значення при аналізі мереж. Вона допомагає виявляти ізольовані групи або оцінювати цілісність системи. Також існують спеціалізовані графи, як-от двочасткові графи, де вершини поділяються на дві окремі множини, і ребра з'єднують лише вершини з різних множин, або мультиграфи, що допускають наявність кількох ребер між однією й тією ж парою вершин. Кожен із цих типів розроблений для вирішення конкретних завдань моделювання. Така різноманітність підкреслює потужність і гнучкість графів як універсального інструменту для розв'язання складних обчислювальних задач у різних галузях.

Графи є універсальними DS, які знаходять широке застосування як у теоретичних дослідженнях, так і в різноманітних практичних обчислювальних завданнях. Завдяки їх здатності моделювати складні взаємозв'язки між об'єктами, вони є незамінними в багатьох сферах.

Однією з сфер застосування графів є соціальні мережі та системи рекомендацій. На платформах типу Facebook або LinkedIn користувачі виступають вершинами, а їхні дружні зв'язки або взаємодії — ребрами. Завдяки цьому можна ефективно аналізувати мережу відносин між людьми. Схожим чином Amazon або Netflix використовують графи, щоб пропонувати користувачам фільми, товари чи послуги на основі їхніх вподобань, історії переглядів і схожості з іншими об'єктами.

Графи відіграють ключову роль у системах навігації та інфраструктури мереж. В алгоритмах маршрутизації, таких як алгоритми Дейкстри або Беллмана-Форда, графи використовуються для знаходження найкоротшого шляху між двома точками, що є основою функціонування картографічних сервісів на кшталт Google Maps. Також графові структури даних є невід'ємними для представлення комп'ютерних мереж, де кожен пристрій виступає вершиною, а комунікаційні зв'язки між ними - ребрами. [19]

Крім того, графи знаходять своє застосування у веб-індексації, електроніці та управлінні проектами. Пошукові системи, як-от Google, широко використовують графи для індексування Інтернету, розглядаючи веб-сторінки як вершини, а гіперпосилання між ними - як ребра, формуючи тим самим величезний орієнтований граф. В електроніці графи застосовуються для представлення схем, де компоненти (резистори, конденсатори, котушки індуктивності) є вершинами, а їхні з'єднання - ребрами. Навіть в управлінні проектами графи корисні для планування, де окремі завдання є вузлами, а ребра позначають залежності між ними, забезпечуючи логічну послідовність виконання робіт.

Деревоподібні структури даних:

Деревоподібна структура даних схожа на зв'язані списки тим, що кожен вузол містить дані і пов'язаний з іншими вузлами. Раніше ми розглядали лінійні структури даних - масиви, зв'язані списки, стеки та черги, де елементи розташовані один за одним у певній послідовності. На відміну від них, дерево дозволяє одному елементу мати кілька «наступних» елементів, через що структура розгалужується в різні боки.

Залежно від типу дерева, його властивості можуть відрізнятися. Наприклад, у бінарному дереві пошуку кожен вузол має не більше двох нащадків, а також дотримується певних правил впорядкування — це робить пошук, вставку і видалення елементів більш ефективними.

Дерева лежать в основі багатьох алгоритмів - обходів, пошуку, сортування, прийняття рішень і побудови індексів. Завдяки своїй гнучкості і ефективності, дерева стали основним інструментом у теорії алгоритмів і практичній реалізації складних задач.

У наступному розділі буде розглянуто окремі види дерев, їх реалізацію на мові програмування Python, а також приклади застосування в телекомунікаційних системах. [26-30]

1.3.3 Відмінності між лінійними та нелінійними структурами даних

Основна відмінність полягає в тому, що елементи лінійних DS зберігаються послідовно, тобто один за одним. До типових прикладів таких структур належать масиви, зв'язані списки, стеки та черги. Натомість, елементи нелінійних DS розташовані ієрархічно або у взаємопов'язаний спосіб, без строгої послідовності. Найвідомішими прикладами тут є дерева та графи.

Різниця в організації елементів безпосередньо впливає на процес їхнього обходу. У лінійних структурах обхід є послідовним. Це спрощує операції, але обмежує можливості представлення складних взаємозв'язків. Обхід

нелінійних структур часто буває складнішим, адже існує багато різних шляхів, якими можна пройти по елементах. Це додає гнучкості, але водночас ускладнює процес їхнього перегляду.

У лінійних структурах даних пам'ять зазвичай виділяється безперервно, що означає розміщення елементів у суміжних комірках пам'яті. Натомість у нелінійних структурах пам'ять виділяється динамічно — елементи можуть розташовуватись у різних ділянках пам'яті, а зв'язки між ними реалізуються через вказівники або посилання. Такий підхід забезпечує більшу гнучкість в управлінні даними та їхній організації.

Ще однією суттєвою відмінністю є кількість наступників, які може мати кожен елемент. Для лінійних структур характерно, що кожен елемент має не більше одного наступника (окрім останнього), що відображає їхню послідовну, одновимірну природу. Натомість у нелінійних структурах один елемент може мати множину наступників, що дозволяє формувати складні розгалужені або мережеві структури

Завдяки своїм властивостям, лінійні DS підходять для завдань, де потрібен послідовний доступ і чіткий порядок елементів — наприклад, коли потрібно просто і ефективно працювати з впорядкованими даними. Водночас нелінійні DS незамінні, коли потрібно відобразити та опрацювати складні взаємозв'язки — як це трапляється у соціальних мережах, організаційних схемах чи файлових системах, де зв'язки між елементами можуть бути багатовимірними і розгалуженими. [8]

1.4 Алгоритми в структурах даних

В області комп'ютерних наук алгоритм - це чітко визначена і кінцева послідовність кроків, спрямована на розв'язання конкретної задачі або виконання обчислень. У поєднанні з DS алгоритми визначають способи обробки інформації: вставку, видалення, пошук, оновлення чи обхід елементів.

Ефективність роботи будь-якої структури даних залежить від алгоритмів, що з нею взаємодіють.

Різні типи DS мають власні цілі та особливості, тому правильний вибір алгоритму для виконання операцій над конкретною структурою є ключовим фактором продуктивності. Одне й те саме завдання — наприклад, пошук значення або сортування — може бути реалізоване кількома алгоритмами, кожен із яких має свої переваги та компроміси щодо швидкості, споживання пам'яті та складності.

Оптимальні алгоритми забезпечують ефективність обробки великих обсягів даних, що є надзвичайно важливим у системах реального часу або у сферах із високими вимогами до продуктивності, таких як телекомунікації, фінанси та штучний інтелект. Завдяки цьому, вони гарантують стабільну та швидку роботу навіть у найскладніших умовах.

Загальні операції та пов'язані з ними алгоритми:

Однією з основних операцій є обхід, який передбачає постійний перегляд кожного елемента у DS. Це гарантує, що ні один елемент не буде пропущений під час обробки. Наприклад, для деревоподібних структур існують спеціальні алгоритми обходу, такі як in-order, pre-order та post-order, які є дуже важливими для завдань обчислення виразів або ієрархічної обробки інформації.

Операції вставки та видалення ефективні в додаванні нових елементів або видаленні існуючих із DS. У збалансованих деревоподібних структурах ці дії часто вимагають додаткових кроків, наприклад, обертань, для збереження їхніх структурних властивостей та забезпечення оптимальної продуктивності.

Для знаходження потрібної інформації використовується операція пошуку. Алгоритми пошуку варіюються від простого лінійного пошуку, який послідовно перевіряє кожен елемент, до більш складних та швидких методів, таких як бінарний пошук (для відсортованих даних) або пошук на основі хешування. [14]

Сортування - це процес впорядкування елементів у DS за певним порядком, за зростанням, чи за спаданням. Існує такі алгоритми сортування як, швидке сортування, сортування злиттям та купчасте сортування. Вибір певного алгоритму для масивів або списків залежить від того, як швидко він працює і скільки пам'яті використовує.

Для деревоподібних структур існує операція балансування. Алгоритми балансування є специфічними для цих структур і їхнє завдання полягає в тому, щоб підтримувати оптимальну висоту дерева. Це гарантує ефективний час для операцій пошуку, вставки та видалення, оскільки розбалансоване дерево може втратити свої переваги у продуктивності. [1, 7, 28-30]

Висновки

1) У першому розділі було розкрито історичні дані розвитку структур даних та алгоритмів, що дозволило простежити етапи еволюції від простих до більш складних підходів до організації інформації. Було визначено основні поняття структур даних та здійснено їхню класифікацію на лінійні та нелінійні структури, що є фундаментом для подальшого аналізу. Розуміння цих класифікацій дозволяє краще орієнтуватися в особливостях кожного типу та визначати їх переваги й недоліки.

2) Через зростання складності інженерних завдань і збільшенням обсягів даних виникає потреба у використанні більш просунутих структур, таких як дерева та графи, що дають змогу моделювати складні взаємозв'язки між об'єктами. Систематизація структур на лінійні та нелінійні є критично важливою для правильного вибору оптимальних методів обробки інформації, оскільки кожен тип має власні сфери ефективного застосування.

3) Алгоритми відіграють ключову роль у забезпеченні ефективного зберігання, пошуку та оброблення інформації у різних типах DS. У цьому розділі було також окреслено загальні риси деревоподібних структур як одного з представників нелінійних структур, що слугуватиме основою для детального

аналізу їхніх характеристик та прикладного значення у наступному розділі роботи.

РОЗДІЛ 2

ДЕРЕВОПОДІБНІ СТРУКТУРИ ДАНИХ

2.1 Загальні поняття про деревоподібні структури даних

Деревоподібна структура даних (англ. Tree Data Structure, TDS) - це нелінійна, ациклічна, ієрархічна організація даних, яка складається з вузлів, що з'єднані ребрами. У деревах дані організовуються таким чином, що відображає природні ієрархії. Кожен елемент, або вузол може мати одного або кількох нащадків, але не може мати більше одного батька. Завдяки своїй будові дерева чудово підходять для моделювання різних ієрархічних залежностей у даних, що робить їх незамінними в багатьох сферах комп'ютерної науки. Вони допомагають не лише краще організувати інформацію, а й значно спрощують і прискорюють пошук потрібних даних - особливо завдяки ефективним алгоритмам обходу, таким як обхід у порядку, перед- чи післяпорядковий обхід.

Кожен вузол дерева зберігає певну інформацію (дані) та має посилання на свої дочірні вузли. Якщо у вузла немає дочірніх - його називають листком. Структура дерева може бути дуже різною за формою та розміром, і саме це впливає на те, наскільки швидко працюють алгоритми пошуку, вставки чи видалення даних. Однією з головних переваг дерев є те, що вони дозволяють обробляти дані доволі швидко - у середньому за логарифмічний час. Завдяки цьому дерева дуже зручні для задач, де критично важлива швидкість роботи з інформацією.

Для подальшого вивчення важливо ознайомитися з основними термінами та поняттями, що пов'язані з TDS: корінь (англ. root), вузол (англ. node), ребро (англ. edge), листок (англ. leaf), рівень (англ. level) і глибина (англ. depth). Розуміння цих понять дає основи для побудови і аналізу більш складних структур, таких як бінарні дерева, дерева пошуку та інші різновиди, які будуть

детально розглянуті у наступних підпунктах цього розділу. Рисунок 2.1 показує ієрархічну природу деревоподібних структур даних (TDS).

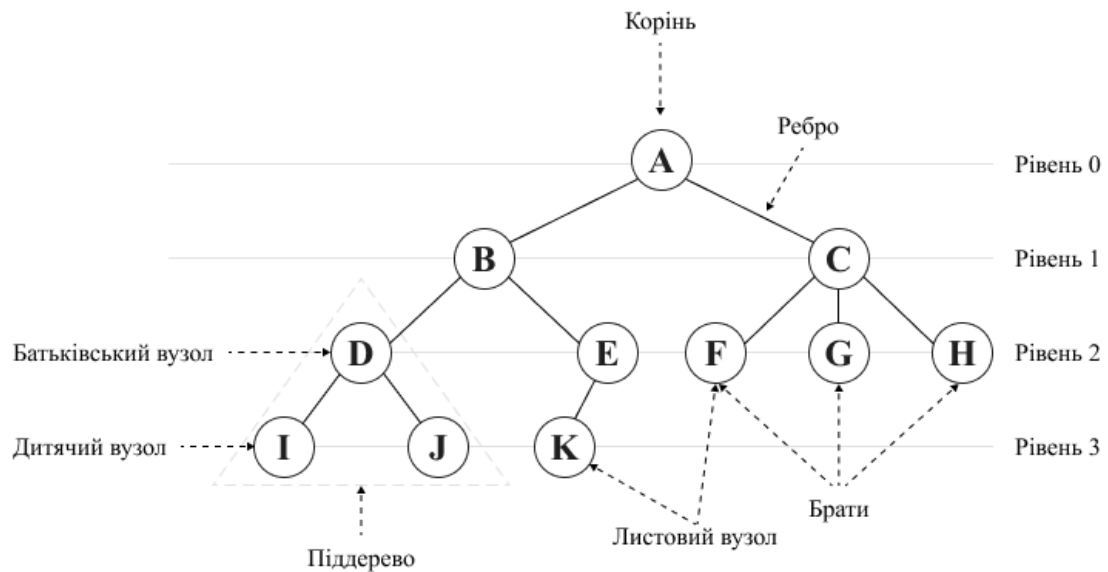


Рис. 2.1 Ієрархічна природа деревоподібних структур даних

Основні елементи TDS:

- Батьківський вузол (англ. Parent Node): вузол, який є попередником вузла, називається батьківським вузлом цього вузла. {B} є батьківською вершиною для {D, E}.
- Дочірній вузол (англ. Child Node): вузол, який є нащадком вузла, називається дочірнім вузлом цього вузла. {D, E} є дочірніми вершинами вершини {B}.
- Кореневий вузол (англ. Root Node): вершина, яка знаходиться на верхівці дерева або не має батьківського вузла, називається кореневим вузлом. {A} є кореневою вершиною дерева. Кожне непорожнє дерево повинно мати лише один кореневий вузол і єдиний шлях від нього до всіх інших вузлів.

- Листяний вузол або зовнішній вузол (англ. Leaf Node or External Node): вершини, які не мають дочірніх вершин, називаються листовими. $\{I, J, K, F, G, H\}$ є листовими вузлами дерева.
- Предок вершини (англ. Ancestor of a Node): будь-які вузли, які розташовані на шляху від кореня до певної вершини, називаються її предками. $\{A, B\}$ є вершинами-предками вершини $\{E\}$.
- Нащадок (англ. Descendant): вузол x вважається нащадком вузла y тоді і тільки тоді, коли y є предком x .
- Рідний брат (англ. Sibling): вузли, які мають спільний батьківський вузол, називаються братами та сестрами. $\{D, E\}$ називаються братами та сестрами.
- Рівень вершини (англ. Level of a node): кількість ребер на шляху від кореневої вершини до цієї безпосередньої вершини. Корінь дерева має рівень 0.
- Внутрішня вершина (англ. Internal node): будь-який вузол, який має хоча б одного нащадка, називається внутрішнім вузлом.
- Сусід вузла (англ. Neighbour of a Node): сусідами вузла називають його безпосередніх батьківських або дочірніх вузлів.
- Піддерево (англ. Subtree): піддерево складається з будь-якої вершини з усіма її нащадками. [10]

Представлення TDS:

Дерево починається з кореневого вузла, від якого можуть відгалужуватися нуль або більше піддерев, позначених як $T_1, T_2, \dots, T_k$. Кожне таке піддерево з'єднується з коренем основного дерева за допомогою ребра. Піддерево певного вузла X охоплює всі вузли, для яких X виступає предком. На Рисунку 2.2 можна побачити представлення деревоподібної структури даних (TDS).

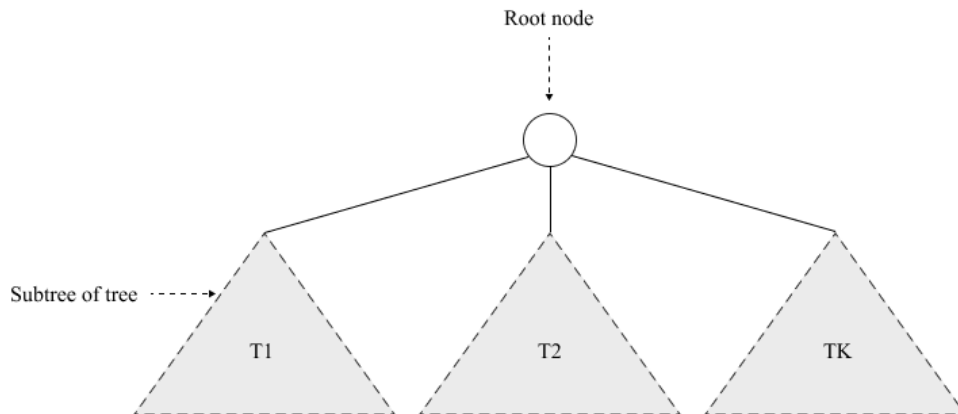


Рис. 2.2 Представлення деревоподібних структур даних

Представлення вузла в TDS на мові програмування Python:

Дерево можна представити як набір вузлів, де кожен зберігає певні дані (інформацію) та має посилання на інші вузли. По суті, вся структура дерева складається саме з цих взаємопов'язаних елементів. У програмуванні кожен вузол зазвичай реалізується за допомогою спеціальних конструкцій - наприклад, класів або структур. Це дозволяє описувати те, що саме вузол буде зберігати і як він буде пов'язаний з іншими вузлами. Нижче наведено простий приклад реалізації вузла (Node) за допомогою класу в мові програмування Python. Цей приклад коду показує, як можна інкапсулювати дані вузла та посилання на його дочірні елементи, при цьому створюючи базовий будівельний блок для будь-якої TDS:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.children = []
```

Детальний опис елементів класу Node:

Клас Node є основним елементом для побудови TDS. Цей клас оголошується за допомогою `class Node:`, що створює шаблон для об'єктів, які

будуть представляти окремі вузли в дереві. Конструктор класу, `def __init__(self, data):`, автоматично запускається кожного разу при створенні нового вузла. У цьому методі `self` посилається на створюваний об'єкт, що дозволяє отримати доступ до його атрибутів, а `data` - це параметр, який призначений для зберігання інформації, яка пов'язана з конкретним вузлом.

Наступні два рядки коду ініціалізують ключові атрибути вузла. `self.data = data` присвоює передане значення `data` атрибуту `data` поточного екземпляра вузла, саме це забезпечує унікальність даних для кожного вузла. Водночас `self.children = []` ініціалізує атрибут `children` як порожній список. Він призначений для зберігання посилань на дочірні вузли поточного вузла. Використання списку дозволяє вузлу мати довільну кількість нащадків, що є характерним для дерев загального призначення. Спочатку список `children` порожній, оскільки щойно створений вузол ще не має дітей; вони додаватимуться до цього списку в процесі побудови дерева. [9, 10, 12]

2.1.1 Властивості деревоподібних структур даних

Кількість ребер (англ. edge):

Ребро - це зв'язок між двома вузлами дерева. Якщо в дереві кількість вузлів дорівнює N , то кількість ребер буде дорівнювати $(N - 1)$. Треба зауважити, що існує один унікальний шлях від будь-якого вузла до будь-якого вузла в дереві.

Глибина вузла (англ. depth of a node):

Глибина вузла визначається як відстань від кореня дерева до цього вузла. Кожне ребро дерева на цьому шляху додає лише одну одиницю до довжини. Також, глибину вузла можна розглядати як кількість ребер на шляху від кореня до заданого вузла дерева.

Висота вузла (англ. height of a node):

Висотою вузла називають довжину найдовшого шляху від цього вузла дерева до одного з його листових вузлів, які є його нащадками.

Висота дерева (англ. height of a tree):

Висота дерева представляє собою довжину найдовшого шляху від кореневого вузла дерева до будь-якого з його листових вузлів.

Ступінь вузла (англ. degree of a node):

Ступенем вузла називається кількість безпосередніх нащадків або кількість піддерев, які виходять з цього вузла. Листовий вузол, який не має дочірніх елементів, ступінь буде завжди дорівнювати 0. Сам ступінь самого дерева визначається як максимальний ступінь серед усіх вузлів у цьому дереві.
[13]

2.2 Бінарні структури даних

Бінарне дерево (англ. Binary Tree, BT) - це особливий вид ієрархічної структури даних, яка не є лінійною. Ключовою характеристикою BT є те, що кожен вузол може мати не більше двох нащадків: один, який називається лівим дочірнім вузлом, і інший, який називається правим дочірнім вузлом. Найвищий вузол у бінарному дереві називається коренем, а вузли, які розташовані нижче, називаються листям.

Структура представлення BT:

У BT кожен вузол складається з трьох частин:

- Дані (англ. Data) - це інформація, яка зберігається у вузлі.
- Показчик на лівого нащадка (англ. Left child) - він вказує на ліве піддерево, а саме, на вузол, який є лівим нащадком поточного.
- Показчик на правого нащадка (англ. Right child) - він вказує на праве піддерево, а саме, на вузол, який є правим нащадком поточного.

Кореневий вузол являє собою початкову точку дерева та не має батьківського вузла. Кожен вузол дерева може мати не більше двох нащадків: лівого та правого показника, що й є визначенням BT.

На Рисунку 2.3 показано приклад простої структури ВТ. Кореневий вузол має лівого нащадка та правого нащадка. Кожен з цих вузлів іще має по два вказівники на потенційні нащадки, які в даному випадку є порожніми (тобто рівними NULL). Це означає, що вони не мають подальших дочірніх вузлів і є листковими. [13, 15]

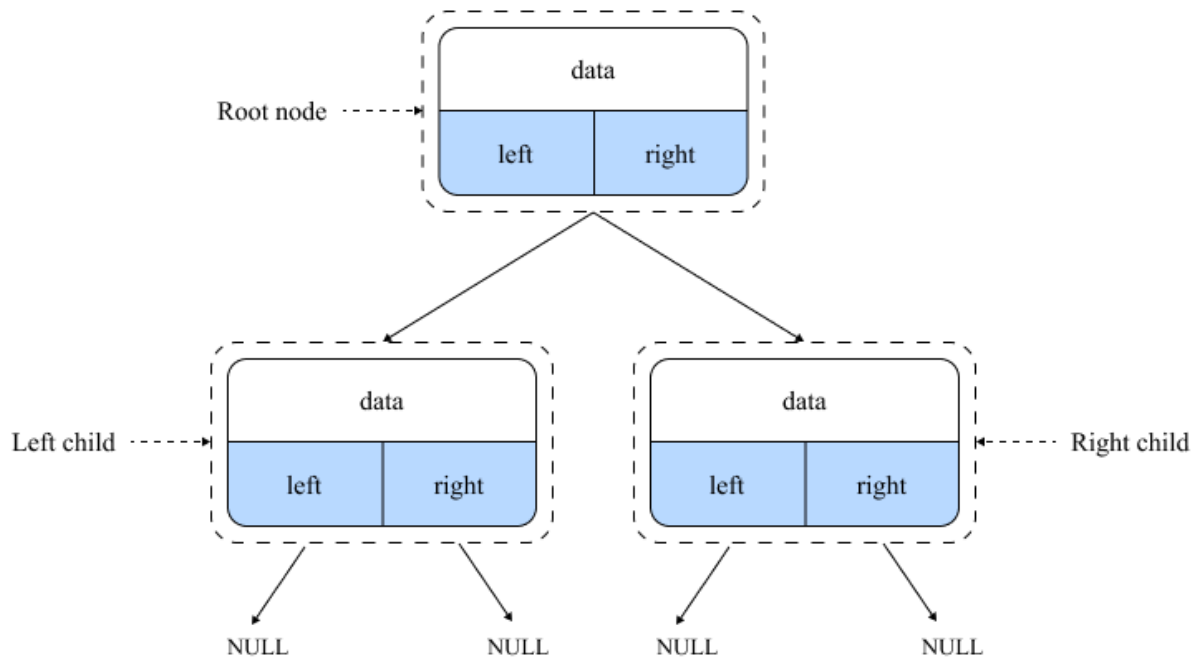


Рис. 2.3 Схематичне представлення бінарного дерева (BT)

Цей рисунок показує типову реалізацію вузла у програмуванні, де вузол зберігає дані та має посилання на лівого і правого нащадка. Ця структура дозволяє ефективно реалізовувати рекурсивні алгоритми обходу, пошуку та вставки елементів у дерево. [14]

Ключова термінологія у ВТ:

- Вузол (анг. Node): Основний елемент ВТ, який містить дані та посилання на лівий та правий вузол.

- Кореневий вузол (англ. Root): Найвищий вузол у дереві, з якого починається структура даних. Він не має батьківського вузла і являє собою початкову точку для всіх інших вузлів у дереві.
- Батьківський вузол (англ. Parent Node): Вузол, який має хоча б одного дочірнього вузла. У ВТ кожен вузол може мати не більше двох дітей.
- Дочірній вузол (англ. Child Node): Вузол, яких є нащадком іншого вузла (свого "батька").
- Листковий вузол (англ. Leaf Node): Вузол, який не має жодного дочірнього вузла, тобто обидва його посилання на дітей дорівнюють NULL.
- Внутрішній вузол (англ. Internal Node): Вузол, що має принаймні одного нащадка. Всі вузли дерева, крім листкових, є внутрішніми.
- Глибина вузла (англ. Depth of a Node): Кількість ребер (зв'язків) від даного вузла до кореневого. Глибина кореневого вузла дорівнює нулю.
- Висота бінарного дерева (англ. Height of a Binary Tree): Кількість вузлів у найдовшому шляху від найглибшого листкового вузла до кореня дерева.

На Рисунку 2.4 показано термінологію бінарного дерева.

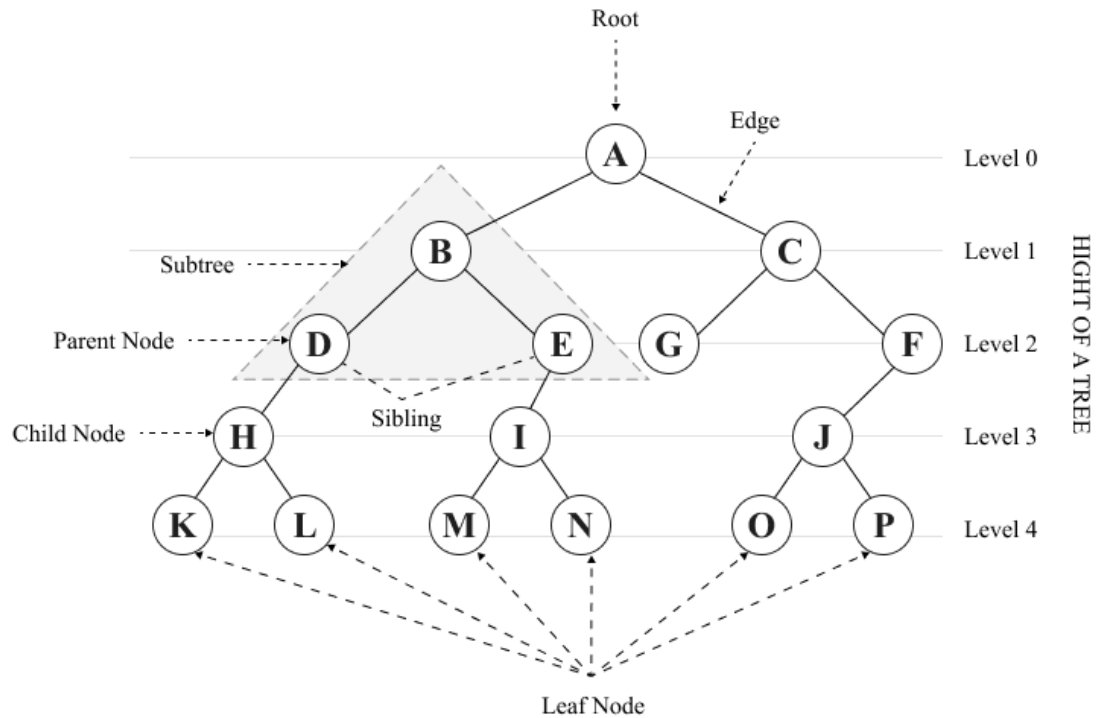


Рис. 2.4 Термінологія бінарного дерева (BT)

Представлення вузла бінарного дерева мовою програмування Python:

Наведений нижче фрагмент коду Python показує типову реалізацію вузла (Node) для BT. Цей клас є основним будівельним блоком, який дозволяє створювати ієрархічні структури, де кожен вузол може мати щонайбільше двох "дітей": лівого та правого.

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key
```

2.2.1 Властивості бінарного дерева:

Під час аналізу та реалізації алгоритмів, які працюють з TDS, головне місце займає вивчення властивостей BT. Завдяки розумінню цих властивостей

можна ефективно оцінювати складність операцій, оптимізувати DS та забезпечувати коректну побудову моделей у практичних застосуваннях, зокрема в галузі телекомунікацій. У Таблиці 2.1 наведено ключові математичні та логічні характеристики бінарного дерева, які є важливими для подальших розрахунків та проєктування.

Таблиця 2.1

Властивості ВТ.

№	Властивість	Формулювання
1	Максимальна кількість вузлів на рівні $*L*$	2^L
2	Максимальна кількість вузлів у бінарному дереві висоти $*H*$	$2^H - 1$
3	Кількість листкових вузлів у дереві	Кількість вузлів з двома нащадками + 1
4	Мінімальна можлива висота дерева з $*N*$ вузлами	$\log_2(N + 1)$
5	Мінімальна кількість рівнів у дереві з $*L*$ листками	$\lceil \log_2 L \rceil + 1$

Зазначені вище властивості мають чітке математичне підґрунтя, яке допомагає ефективно оцінювати характеристики дерев під час їх створення. Далі наведені детальні пояснення до кожної з цих властивостей.

Максимальна кількість вузлів на рівні та загальна кількість вузлів:

На кожному рівні l бінарне дерево може містити щонайбільше 2^l вузлів. При цьому корінь дерева знаходиться на рівні 0. Цю властивість легко довести індукцією: якщо на рівні l є 2^l вузлів, то на наступному рівні $l+1$ їх буде не більше $2 \times 2^l = 2^{(l+1)}$. Виходячи з цього, ВТ висотою h (де висота - це довжина найдовшого шляху від кореня до листа, а корінь має висоту 0) може мати максимум $2^{(h+1)} - 1$ вузлів. Ця формула отримана шляхом підсумовування вузлів на всіх рівнях: $1 + 2 + 4 + \dots + 2^h$.

Мінімальна висота та кількість рівнів:

Для дерева з N вузлами, мінімально можлива висота становить $\lceil \log_2 N \rceil$. Це випливає з того, що дерево з висотою h може містити не більше $2^{(h+1)}-1$ вузлів, звідки $N \leq 2^{(h+1)}-1$, що після перетворень дає $h \geq \log_2(N+1)-1$, або ж $h \geq \lceil \log_2 N \rceil$. Аналогічно, ВТ з L листовими вузлами повинно мати принаймні $\lceil \log_2 L \rceil$ рівнів. Це пояснюється тим, що максимальна кількість листя досягається при повному заповненні всіх рівнів, і на певному рівні l може бути щонайбільше 2^l листів.

Взаємозв'язок між внутрішніми та листовими вузлами:

У повному ВТ, де кожен вузол має або нуль, або два дочірні елементи, кількість листових вузлів (L) завжди на одиницю більша, ніж кількість внутрішніх вузлів (T), які мають двох дітей. Ця властивість виражається формулою $L = T + 1$. Доведення базується на тому, що повне ВТ висоти h має $2^{(h+1)}-1$ вузлів, з яких 2^h є листям, а 2^h-1 - внутрішніми вузлами.

Загальна кількість ребер:

Будь-яке непорожнє ВТ з N вузлами завжди має рівно $N-1$ ребро. Це пояснюється тим, що кожен вузол у дереві, за винятком кореня, має рівно одного батька. Оскільки кожен зв'язок "батько-дитина" представляє одне ребро, а таких зв'язків $N-1$ (бо корінь не має батька), то і кількість ребер дорівнює $N-1$. [11-18]

2.2.2 Операції над бінарним деревом

Алгоритм пошуку в глибину (англ. Depth-First Search, DFS) - це спосіб обходу дерева або графа, при якому програма намагається зануритися якомога глибше в одну гілку, перш ніж переходити до інших. Коли гілка закінчується або вузол уже відвіданий, алгоритм повертається назад і починає досліджувати інші можливі шляхи. Зазвичай такий підхід реалізують за допомогою рекурсії, що дозволяє зручно стежити за напрямком обходу.

Прямий обхід бінарного дерева (англ. Preorder Traversal of Binary Tree):

Прямий обхід дерева відбувається за принципом "корінь, ліве піддерево, праве піддерево". Це означає, що спершу ми відвідуємо (обробляємо) сам поточний вузол, потім рекурсивно заглиблюємося в його ліве піддерево, і лише після цього переходимо до правого піддерева. Цей обхід є надзвичайно корисним, коли потрібно створити точну копію дерева, оскільки він дозволяє обробляти батьківські вузли до їхніх нащадків. Також прямий обхід застосовується для отримання префіксного (польського) запису математичних виразів, коли оператор стоїть перед операндами.

Інфіксний обхід (англ. In-order Traversal of Binary Tree):

При інфіксному обході послідовність дій така: "ліве піддерево, корінь, праве піддерево". Алгоритм спочатку рекурсивно обходить все ліве піддерево, потім відвідує поточний вузол, і вже після цього переходить до правого піддерева. Найважливіша особливість інфіксного обходу полягає в його здатності сортувати елементи. Цей обхід використовується для генерації звичайного інфіксного запису математичних виразів, де оператор розташований між операндами.

Зворотний обхід (англ. Post-order Traversal of Binary Tree):

Зворотний обхід дотримується послідовності "ліве піддерево, праве піддерево, корінь". У цьому випадку спочатку повністю обходяться всі вузли лівого піддерева, потім усі вузли правого піддерева, і тільки тоді відвідується сам поточний вузол. Такий порядок обходу ідеально підходить для операцій, де необхідно обробити всі нащадки перед самим батьківським вузлом. Класичний приклад - видалення всього дерева з пам'яті, оскільки починаємо з листків і рухаємося вгору, запобігаючи "висячим" посиланням. Також зворотний обхід дозволяє отримати постфіксний (зворотний польський) запис математичних виразів, де оператор йде після операндів. [18,19]

2.3 Бінарні дерева пошуку

У комп'ютерних науках, особливо в області зберігання та обробки структурованої інформації, бінарні дерева пошуку (англ. Binary Search Trees, BST) відіграють важливу роль. Це особливий тип динамічної DS, який дозволяє зберігати елементи таким чином, щоб до них можна було швидко отримати доступ, оновити або видалити.

Бінарне дерево пошуку (BST) - це структура даних, яка застосовується в інформатиці для організації та зберігання даних у відсортованому вигляді. Кожен вузол BST може мати не більше двох дочірніх вузлів - лівий і правий. При цьому значення в лівому дочірньому вузлі завжди менші за значення батьківського вузла, а в правому - більші. Завдяки цій впорядкованій структурі, BST забезпечує ефективне виконання основних операцій, таких як пошук, вставка та видалення елементів.

BST це така структура даних, яка підтримує операції пошуку елемента, мінімального і максимального значення, попередника й наступника, вставку і видалення. Основні операції в BST виконуються за час, який пропорційний його висоті. Для повного бінарного дерева з n вузлами ці операції виконуються за час $\Theta(\lg n)$ у найгіршому випадку. Математичне сподівання висоти побудованого випадковим образом бінарного дерева равно $O(\log n)$, так що всі основні операції над динамічною множиною в такому дереві виконуються в середньому за час $\Theta(\lg n)$.

Однак в реальних умовах не завжди можна забезпечити випадкову структуру дерева, що може призвести до збільшення висоти та погіршення продуктивності. Для того, щоб уникнути цього, використовують спеціальні види дерев, зокрема червоно-чорні дерева, у яких висота завжди залишається $O(\log n)$, що в свою чергу гарантує стабільну ефективність операцій навіть у найгірших випадках. [13]

Ключовою властивістю BST:

Вузли в бінарному дереві пошуку (BST) розташовуються відповідно до чітко визначеного правила впорядкування, що відрізняє цю структуру від звичайного бінарного дерева. Це правило забезпечує впорядкованість елементів, що сприяє підвищенню ефективності операцій пошуку, вставки та видалення даних.

- Ліве піддерево вузла містить лише вузли з ключами, меншими за ключ вузла.
- Праве піддерево вузла містить лише вузли з ключами, більшими за ключ вузла.
- Ліве і праве піддерево також повинні бути бінарними деревами пошуку.
- BST може мати повторювані значення при різних підходах до обробки.

Структура вузла в бінарному дереві пошуку (BST):

- ключ (значення);
- посилання на лівого нащадка;
- посилання на правого нащадка;
- (опціонально) посилання на батьківський вузол.

2.3.1 Алгоритми обробки вузлів у BST

Алгоритм пошуку (англ. Search) в BST:

Ефективний пошук елементів є однією з основних операцій у BST. Пошук заданого елемента, позначеного як X , починається з кореневого вузла і виконується за принципом двійкового пошуку, який використовує впорядковану ієрархію даних у дереві. Це дозволяє швидко звужувати область пошуку і знаходити потрібний елемент з мінімальними затратами часу.

На кожному етапі алгоритму виконується порівняння значення X із ключем поточного вузла. У випадку ідентичності значень, пошук успішно завершується, оскільки шуканий елемент знайдено. Якщо X менше за ключ поточного вузла, подальший обхід спрямовується до лівого піддерева. Таке

рішення обґрунтовується властивістю BST, згідно з якою всі елементи, розташовані у лівому піддереві, мають значення, менші за ключ їхнього батьківського вузла. Відповідно, якщо X більше за ключ поточного вузла, траверс продовжується у правому піддереві, де містяться елементи з більшими значеннями.

Цей ітеративний (або рекурсивний) процес продовжується доти, доки не буде досягнуто одного з двох кінцевих станів. Перший стан - це успішне знаходження елемента X , що підтверджується поверненням логічного значення `True`. Другий стан настає, коли обхід досягає "порожнього" посилання (наприклад, `NULL` або `nullptr`), що свідчить про відсутність шуканого елемента в дереві, і алгоритм повертає `False`. Така стратегія забезпечує логарифмічну складність пошуку в середньому випадку, що робить BST високоефективним для великих обсягів даних. [16]

Приклад покрокового пошуку елемента в BST:

Для ілюстрації принципу дії алгоритму пошуку в бінарному дереві пошуку (BST) розглянемо приклад пошуку ключа $X=6$ у наведеному BST на Рисунку 2.5

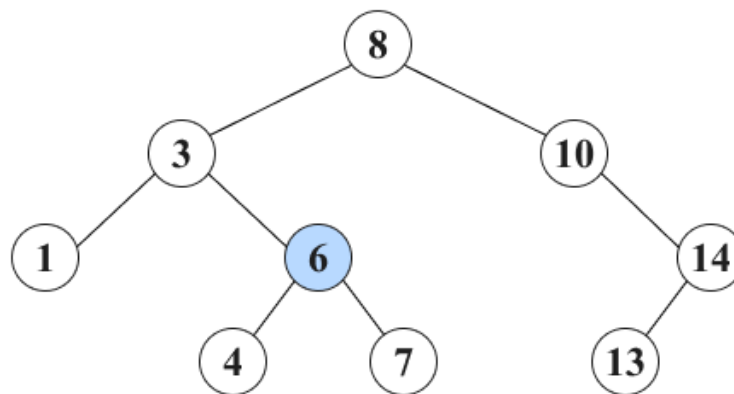


Рис. 2.5 Початкове BST та визначення ключа для пошуку.

Кроком 2 є порівняння з кореневим вузлом та визначення напрямку траверсу. Процедура пошуку ініціюється з кореневого вузла, значення якого дорівнює 8. Здійснюється порівняння шуканого ключа $X=6$ із ключем

кореневого вузла. Оскільки $6 < 8$, згідно з фундаментальною властивістю BST, яка стверджує, що всі елементи лівого піддерева менші за батьківський вузол, алгоритм детермінує необхідність переходу до лівого піддерева відносно кореня. Цей етап ілюструється на Рисунку 2.6, де стрілка вказує на поточний вузол.

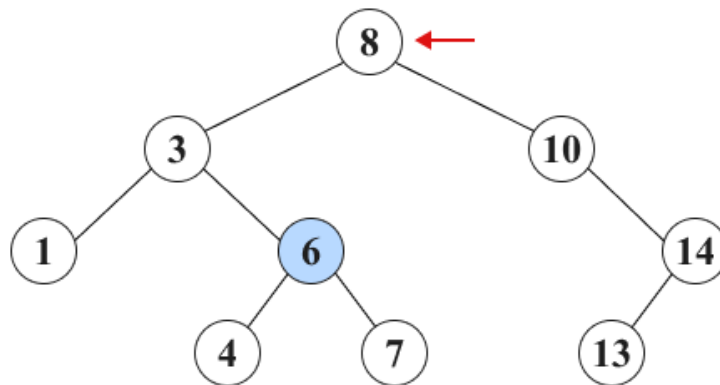


Рис. 2.6 Порівняння ключа з корневим вузлом та вибір лівого піддерева

Кроком 3 є продовження траверсу та подальша редукція простору пошуку. Після переходу до лівого піддерева, поточним вузлом стає вузол зі значенням 3. На цьому етапі повторно виконується порівняння $X=6$ з поточним ключем. Оскільки $6 > 3$, відповідно до аксіоми BST про те, що всі елементи правого піддерева більші за батьківський вузол, подальший пошук спрямовується до правого піддерева вузла 3. Візуалізація цього кроку представлена на Рисунку 2.7.

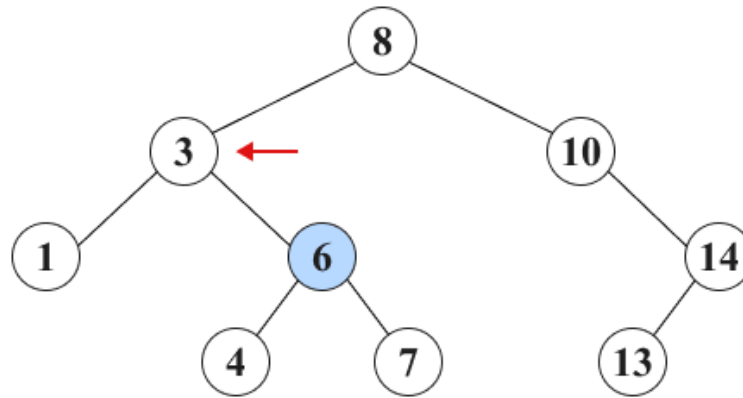


Рис. 2.7 Порівняння ключа з вузлом 3 та вибір правого піддерева

Кроком 4 є завершення пошуку та ідентифікація ключа. Переміщення до правого піддерева від вузла 3 приводить до вузла зі значенням 6. Здійснюється фінальне порівняння шуканого ключа $X=6$ із ключем поточного вузла, яке підтверджує їх еквівалентність. Таким чином, елемент із ключем 6 успішно знайдено в структурі BST. Результат цього етапу показано на Рисунку 2.8, де знайдений вузол виділено.

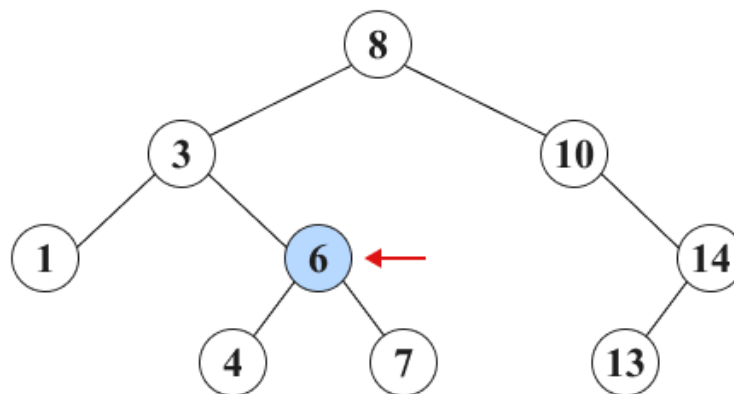


Рис. 2.8 Ідентифікація шуканого ключа 6

Приклад рекурсивної програми для реалізації пошуку в BST на мові програмування Python:

```

class Node:
    def __init__(self, key):
        self.key = key
  
```

```

    self.left = None
    self.right = None

def search(root, key):
    if root is None or root.key == key:
        return root
    if root.key < key:
        return search(root.right, key)
    return search(root.left, key)

root = Node(50)
root.left = Node(30)
root.right = Node(70)
root.left.left = Node(20)
root.left.right = Node(40)
root.right.left = Node(60)
root.right.right = Node(80)

print("Found" if search(root, 19) else "Not Found")
print("Found" if search(root, 80) else "Not Found")

```

Цей код представляє базову реалізацію BST та алгоритму пошуку. Він містить клас Node для вузлів (з key, left, right посиланнями) та рекурсивну функцію search для знаходження елемента за ключем. Функція працює, порівнюючи шуканий ключ з поточним вузлом і рекурсивно переходячи в ліве або праве піддерево. Код включає жорстко закодоване BST для демонстрації, виконуючи пошук як наявного, так і відсутнього елементів.

Алгоритм видалення (ангд. Deletion) в BST:

Операція видалення вузла з BST є однією з найскладніших у реалізації серед базових операцій, тому що вона потребує збереження основної властивості BST - впорядкованості ключів у піддеревах. У процесі видалення

необхідно враховувати структуру дерева та положення вузла, що підлягає видаленню. Від цього залежить, чи буде змінена лише одна гілка дерева, чи потрібно буде перебудувати частину дерева з використанням наступників або попередників. Видалення виконується за трьома основними сценаріями: коли вузол є листом (не має нащадків), коли має одного нащадка (лівого або правого) і коли має двох нащадків. Розглянемо всі три випадки схематично на прикладах. [15]

Приклад 1. Видалення листового вузла в BST:

Даний випадок демонструє найпростіший сценарій операції видалення елемента з BST: усунення кінцевого вузла, тобто вузла, який не має жодних дочірніх компонентів. На лівій частині зображення представлена вихідна конфігурація дерева, де вузол з ідентифікатором 20 визначено як цільовий об'єкт для видалення. Враховуючи, що вузол 20 є листовим (тобто не містить ні лівого, ні правого нащадка), процедура його усунення зводиться до деактивації посилання на нього від його батьківського вузла (вузла 30). Права частина ілюстрації відображає кінцевий результат цієї дії: вузол 20 вилучається зі структури дерева, а відповідний вказівник від вузла 30 отримує значення NULL, ефективно відокремлюючи вилучений елемент від загальної ієрархії. На Рисунку 2.9 схематично показано видалення листового вузла в BST.

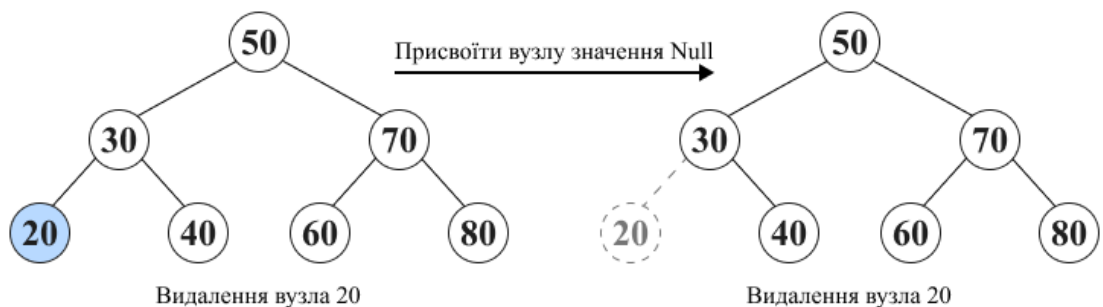


Рис. 2.9 Схема операції видалення листового вузла в BST

Приклад 2: Видалення вузла з одним дочірнім елементом у BST:

На Рисунку 2.10 показано послідовність дій при усуненні вузла, що має лише одного безпосереднього нащадка, зокрема, фокусуючись на вузлі 70, у якого присутній тільки правий дочірній елемент (вузол 80). Цей процес реалізується у два етапи: спершу значення вузла 70 замінюється значенням його єдиного нащадка (80), після чого первинний вузол 80 (який на цьому етапі стає дублікатом) видаляється зі структури дерева, забезпечуючи таким чином збереження його цілісності та специфічних властивостей BST.

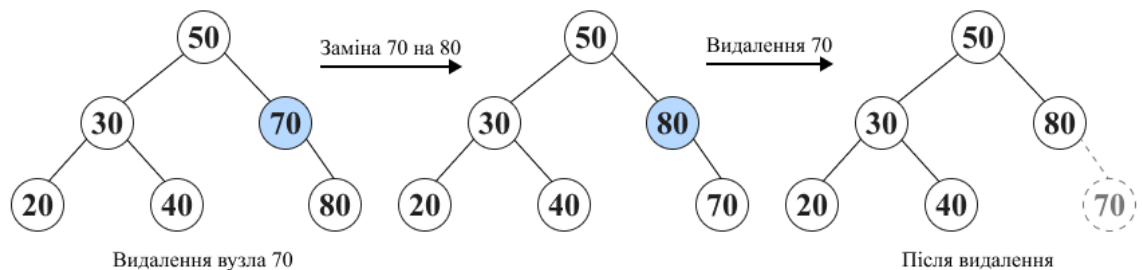


Рис. 2.10 Видалення вузла з одним дочірнім елементом з BST

Приклад 3: Видалення вузла з двома дочірніми елементами у BST:

Цей приклад показує найбільш складний сценарій усунення вузла з BST, а саме коли цільовий вузол має обох нащадків (наприклад, вузол 50). Процедура включає такі етапи: спершу відбувається пошук інфіксного наступника (найменшого елемента в правому піддереві, в даному випадку це 60). Далі здійснюється заміщення значення вузла, що видаляється (50), значенням цього наступника (60). Завершальним кроком є видалення оригінального вузла наступника (60). Такий підхід забезпечує збереження структурної цілісності та властивостей BST. Непарний наступник потрібен лише тоді, коли правий дочірній елемент не є порожнім. У цьому конкретному випадку непарний наступник можна отримати, знайшовши мінімальне значення у правому дочірньому елементі вузла.

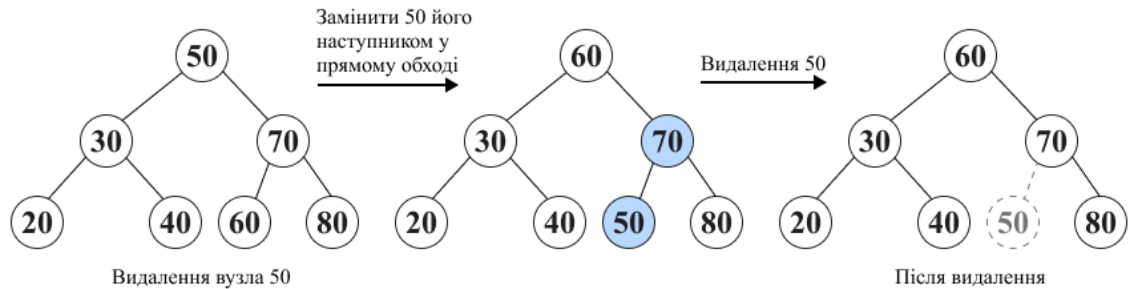


Рис. 2.11 Схема операції видалення вузла з обома дочірніми елементами у BST

Алгоритм вставки (англ. *Insertion*) в BST:

Процедура інтеграції нового елемента у структуру BST завжди передбачає його розміщення у позиції листового вузла, що забезпечує збереження незмінності фундаментальних властивостей цієї деревовидної структури. Імплементация починається з кореневого елемента, після чого відбувається ітеративний пошук оптимального місця для вставки нового ключа. Цей траверс триває до моменту досягнення вузла, який не має відповідного дочірнього елемента, тим самим ідентифікуючи потенційне місце для приєднання.

Після визначення такого термінального вузла, новий елемент з заданим значенням приєднується як його дочірній компонент. Цей процес підпорядковується наступним принципам: ініціалізація поточного розглядуваного вузла (наприклад, *node*) здійснюється з кореневого елемента. Надалі виконується порівняння значення нового ключа із ключем поточного вузла. У випадку, коли значення нового ключа є меншим або рівним значенню поточного вузла, траверс спрямовується до лівого піддерева. Якщо ж значення нового ключа перевищує значення поточного вузла, подальший рух відбувається до правого піддерева. Зазначені кроки повторюються до моменту досягнення листового вузла. Остаточню, новий ключ інтегрується як лівий або правий дочірній елемент цього листового вузла, що визначається результатом

фінального порівняння з його значенням. Розглянемо приклад покрокової вставки елемента в BST: [14]

На Рисунку 2.12 представлено вихідне бінарне дерево пошуку. Нам потрібно вставити новий вузол з ключем 40. Поточний вузол (`currNode`) ініціалізується кореневим вузлом, що містить значення 100.

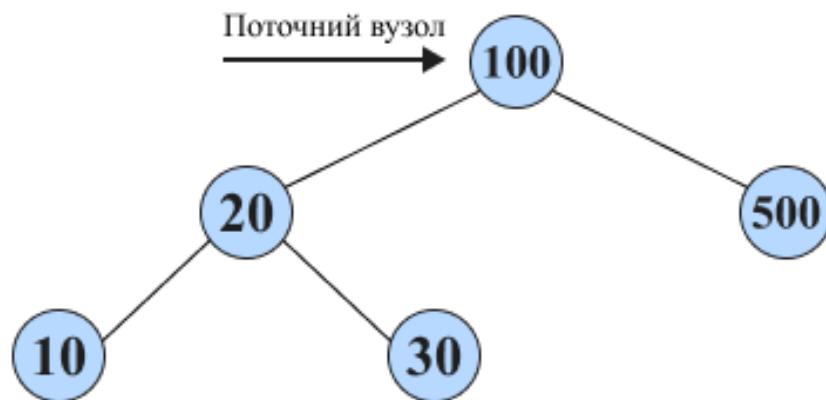


Рис. 2.12 Початковий стан дерева (BST)

Здійснюється порівняння ключа для вставки (40) зі значенням кореневого вузла (100). Оскільки 40 менше за 100, алгоритм спрямовує пошук до лівого піддерева, переміщуючи `currNode` до вузла 20. Цей крок відображено на Рисунку 2.13

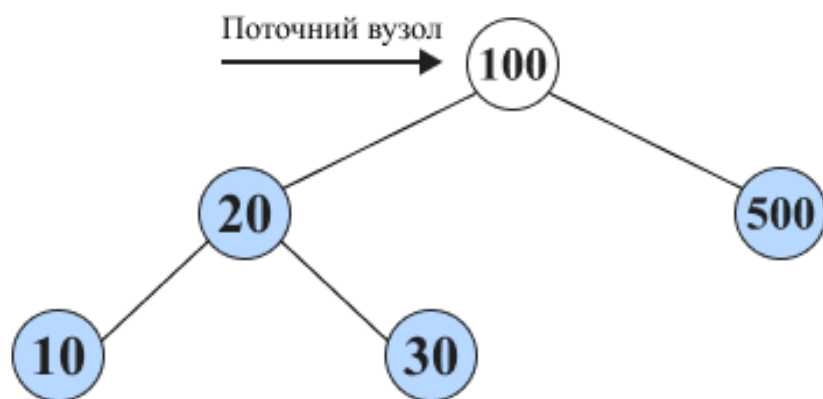


Рис. 2.13 Порівняння ключа з кореневим вузлом (100) та перехід до лівого піддерева

Тепер поточний вузол - 20. Порівнюємо 40 з 20. Оскільки 40 більше за 20, наступний перехід здійснюється до правого піддерева від вузла 20, і `currNode` переміщується до вузла 30. Цей етап ілюструється на Рисунку 2.15

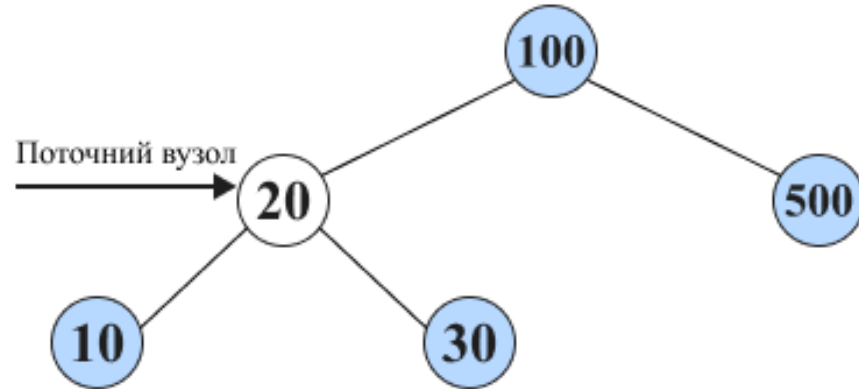


Рис. 2.14 Порівняння ключа з вузлом 20 та перехід до правого піддерева

На даному етапі `currNode` вказує на вузол 30. Знову порівнюємо 40 з 30. Оскільки 40 більше за 30, алгоритм намагається перейти до правого нащадка вузла 30. Це показано на Рисунку 2.15

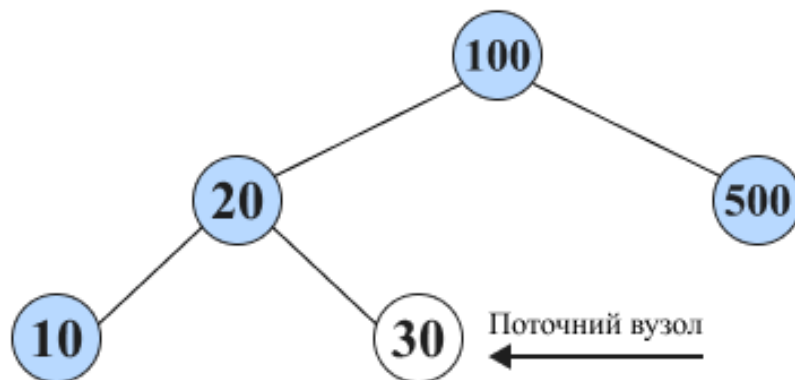


Рис. 2.15 Порівняння ключа з вузлом 30 та пошук місця для вставки

Після переходу від вузла 30 до його правого "нащадка", алгоритм виявляє, що праве посилання є порожнім (NULL). Це означає, що знайдено відповідне місце для вставки нового елемента. Таким чином, новий вузол з

ключем 40 вставляється як правий нащадок вузла 30. Результат цієї операції відображено на Рисунку 2.16

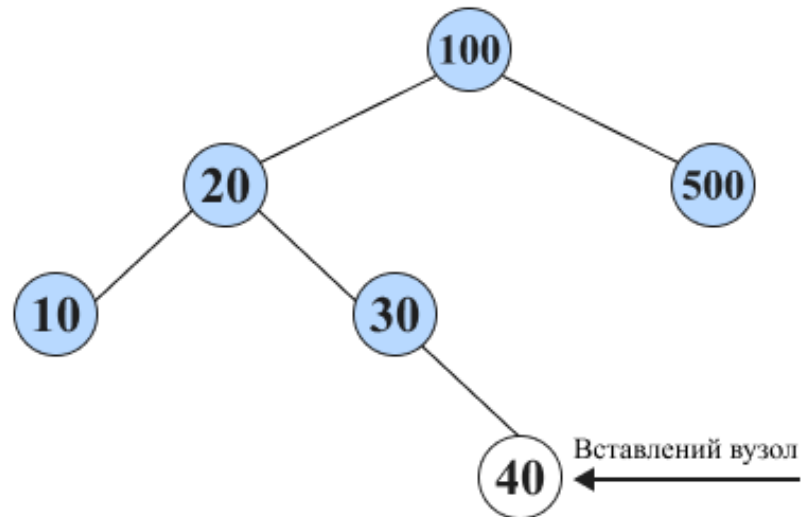


Рис. 2.16 Вставка нового вузла (40) у бінарне дерево пошуку

2.3.2 Застосування BST

Бінарні дерева пошуку — це одна з ключових DS, яка допомагає ефективно працювати з впорядкованими наборами інформації. Головна їхня перевага полягає в тому, що вони підтримують основні операції - пошук, вставку, видалення та навігацію (наприклад, знаходження мінімального або максимального значення) - зі середньою складністю, що зростає приблизно пропорційно логарифму від кількості елементів. Завдяки цьому BST широко застосовуються в різних програмних системах і технологіях обробки даних.

У програмуванні бінарні дерева часто використовують для реалізації асоціативних контейнерів - структур, де дані зберігаються у вигляді пар «ключ-значення». Прикладами таких контейнерів є TreeMap, TreeSet, std::map або std::set. Завдяки впорядкованому розташуванню елементів і швидкому доступу з логарифмічною складністю, ці структури часто вибирають замість хеш-

таблиць, особливо коли потрібно підтримувати сортування або виконувати пошук у певних діапазонах.

Також BST широко застосовуються у завданнях, що пов'язані з обробкою потоків даних. У режимі реального часу часто виникає потреба зберігати значення у впорядкованому вигляді та оперативно відповідати на запити типу: "скільки значень перевищують або не перевищують певний поріг". У таких випадках BST є зручною базою для реалізації ефективної обробки потоку.

У сфері телекомунікацій BST також широко застосовуються. Наприклад:

- У системах маршрутизації дерева дозволяють підтримувати оновлювані таблиці маршрутів, де ключем виступають IP-адреси або маски мереж.
- Під час обробки мережевого трафіку BST можна використовувати для зберігання правил доступу (ACL) у впорядкованому вигляді, що спрощує і пришвидшує перевірку кожного запиту.
- У задачах аналізу активності користувачів дерева можуть використовуватися для зберігання подій з прив'язкою до часу або ідентифікаторів пристроїв, що дозволяє будувати аналітичні запити за часовими інтервалами або умовами.
- При плануванні використання ресурсів каналу - наприклад, у випадку динамічного виділення частот або часових слотів - BST допомагають керувати наявними ресурсами без потреби повного пересканування списку.
- У алгоритмах стиснення або передачі даних бінарні дерева можуть застосовуватись для побудови кодових дерев, де частота символів визначає структуру дерева.

У ситуаціях, де потрібно забезпечити одночасно і впорядкованість, і можливість динамічної зміни набору даних, BST демонструють свою цінність. У телекомунікаційних мережах, де трафік та події виникають непередбачувано, ця властивість є критично важливою.

Для підвищення ефективності, особливо в умовах високих навантажень, часто використовуються самосбалансовані варіації дерев, такі як AVL-дерева або червоно-чорні дерева. Вони гарантують обмеження висоти дерева логарифмом від кількості елементів, що стабілізує час виконання операцій навіть при великому обсязі даних.

Таким чином, BST відіграють важливу роль не лише в комп'ютерних алгоритмах загального призначення, а й у спеціалізованих галузях, таких як телекомунікації, де необхідна висока продуктивність, гнучкість та впорядковане зберігання динамічної інформації. [17]

2.3.3 Аналіз переваг та недоліків бінарних дерев пошуку

Бінарні дерева пошуку становлять один із базових підходів до зберігання впорядкованих даних у вигляді деревоподібної структури, що дозволяє реалізувати ключові операції, такі як пошук, вставка та видалення, з високим рівнем ефективності.

Переваги:

- Висока швидкодія при пошуку: у разі використання збалансованих варіантів BST, таких як AVL-дерева або червоно-чорні дерева, середня та найгірша складність пошуку становить $O(\log n)$, що забезпечує стабільну продуктивність навіть за зростання обсягу даних.
- Природна впорядкованість елементів: структура дерева дозволяє підтримувати всі значення у відсортованому порядку, що, зокрема, полегшує реалізацію операцій на кшталт знаходження наступного або попереднього за значенням вузла.
- Гнучкість у модифікації даних: дерево підтримує динамічну зміну вмісту - додавання та видалення елементів можуть виконуватися з логарифмічною складністю за умови підтримки балансу.

- Підтримка мінімальних і максимальних значень: завдяки структурі вузлів можливо ефективно знаходити як найменший, так і найбільший елемент без необхідності повного обходу дерева.
- Можливість побудови пріоритетних черг з двома напрямками: BST надає змогу реалізувати черги з підтримкою одночасного доступу до обох кінців - максимуму та мінімуму, на відміну від класичної бінарної купи.

Недоліки:

- Відсутність автоматичного балансування: звичайні BST не гарантують підтримку збалансованої форми дерева, що у найгіршому випадку призводить до перетворення дерева на лінійну структуру зі складністю $O(n)O(n)O(n)$.
- Збільшене використання пам'яті: кожен вузол зберігає додаткові вказівники на лівого та правого нащадків, що може створити помітне навантаження на оперативну пам'ять при роботі з великим обсягом даних.
- Обмеження масштабованості: при значному збільшенні розміру вхідних даних звичайні BST без додаткових механізмів оптимізації можуть виявитися менш ефективними, ніж спеціалізовані структури, наприклад, B-дерева.
- Невеликий набір базових операцій: функціональні можливості BST обмежуються стандартними маніпуляціями з вузлами, що знижує універсальність порівняно з іншими DS, такими як хеш-таблиці або сегментні дерева.

2.4 Дерева рішень

Дерева рішень (англ. Decision Tree, DT) — це метод керованого навчання, який не припускає певної форми розподілу даних і використовується для задач класифікації та регресії. Його мета - побудувати модель, яка прогнозує

значення цільової змінної, опираючись на прості правила прийняття рішень, сформовані на основі особливостей вхідних даних.

Дерева рішень базуються на ідеї поступового розбиття вхідних даних на менші підгрупи, що допомагає знаходити найкращі правила для класифікації або прогнозування. Їх можна уявити як спосіб приблизного опису складних залежностей крок за кроком. Ці моделі широко використовують для різних задач - від класифікації та регресії до підтримки прийняття рішень у таких галузях, як телекомунікації, медицина, фінанси, виробництво та інші.

ДТ має ієрархічну структуру дерева, що починається з одного головного питання на вершині, яке називається вузлом, і далі розгалужується на різні можливі результати:

- Кореневий вузол (англ. Root Node) - це вузол, з якого починається дерево. Цей вузол представляє повну множину вхідних даних.
- Гілки (англ. Branches) - вони представляють собою варіанти результатів тестів, які ведуть до наступних вузлів.
- Внутрішні вузли (англ. Internal Nodes) - це точки, де рішення приймаються на основі вхідних даних.
- Листяні вузли (англ. Leaf Nodes) - це кінцеві вузли дерева, які представляють кінцеві результати або прогнозовані значення.

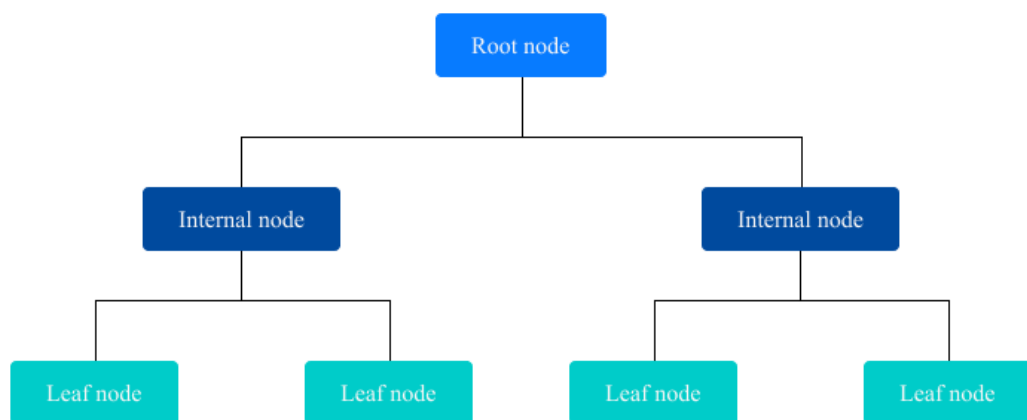


Рис. 2.17 Структура дерева прийняття рішень

У процесі класифікації, для кожного прикладу (рядка вхідних даних) здійснюється послідовний перехід від кореневого вузла до одного з листяних вузлів відповідно до значень атрибутів.

Такі структури також сприяють процесу прийняття рішень, надаючи наочне уявлення про потенційні результати. Це дозволяє швидко аналізувати та зіставляти різні "розгалуження" або варіанти, щоб визначити найбільш ефективний шлях дій. [20]

2.4.1 Типи дерев рішень

DS головним чином розрізняють за двома основними типами, що залежать від характеру прогнозованої змінної: це дерева класифікації та дерева регресії.

Класифікаційні дерева (англ. Classification Trees):

Такі дерева використовуються у тоді, коли цільова змінна є дискретною, тобто такою, що належить до обмеженого набору категорій. Метою дерева є побудова послідовності логічних правил, які дозволяють віднести об'єкт до одного з класів. У кожному вузлі дерева виконується перевірка умови, яка ділить вхідні дані на підмножини, що мають найбільшу гомогенність відносно класу. Типовим прикладом застосування класифікаційного дерева є фільтрація електронної пошти, де алгоритм приймає рішення, чи є повідомлення "спамом" чи "не спамом", ґрунтуючись на таких характеристиках як наявність певних ключових слів, відправник або структура листа.

Регресійні дерева (англ. Regression Trees):

На відміну від класифікаційних, застосовуються у випадках, коли цільова змінна є неперервною та набуває числових значень. Такі дерева поділяють простір даних на області, в межах яких модель прогнозує значення на основі середнього або іншого статистичного критерію. Наприклад, регресійне дерево може бути використано для оцінки ринкової вартості нерухомості, враховуючи такі фактори, як площа, розташування, рік побудови,

кількість кімнат та інші параметри. Кожне розгалуження в дереві відповідає значущому параметру, який найбільше зменшує помилку прогнозу, а кінцеві вузли містять числові оцінки, які є передбачуваним результатом. [24, 25]

2.4.2 Алгоритми побудови дерева рішень

Побудова ефективних дерев рішень є центральним завданням у галузі машинного навчання. Саме для цього було розроблено низку спеціалізованих алгоритмів.

Ці алгоритми базуються на жадібній стратегії пошуку (англ. Greedy Search Strategy). Це означає, що на кожному вузлі дерева вони приймають локально оптимальне рішення - обирають той атрибут (та порогове значення для неперервних атрибутів), який на даний момент найкраще розділяє дані згідно з обраним критерієм (інформаційний приріст, коефіцієнт приросту, індекс Джині, MSE). Хоча це не гарантує глобально оптимального дерева, такий підхід є обчислювально ефективним і на практиці дає хороші результати.

Нижче розглянуто найбільш відомі алгоритми:

ID3 (Iterative Dichotomiser 3):

Алгоритм ID3, розроблений Россом Квінланом, є одним із перших і найпростіших алгоритмів для побудови дерев рішень. Його ключовою особливістю є використання критерію інформаційного приросту (англ. Information Gain) для вибору найкращого атрибута на кожному вузлі.

Алгоритм ID3 створює дерево рішень шляхом рекурсивного розгалуження, починаючи від кореня. На кожному кроці він аналізує доступні дані, щоб визначити, який атрибут забезпечить найбільше зменшення невизначеності - це поняття відоме як інформаційний приріст. Саме цей атрибут, що максимізує ефективність поділу, обирається для формування поточного вузла дерева.

Далі, вихідний набір даних розбивається на менші підмножини, кожна з яких відповідає конкретному значенню обраного атрибута. Цей процес

повторюється для кожної новоствореної підмножини. Ітерації тривають доти, доки всі елементи в певній підмножині не будуть класифіковані як належні до одного класу, або ж поки всі атрибути, доступні для розгалуження, не будуть вичерпані.

Суть інформаційного приросту лежить в концепції ентропії, яка слугує мірою хаотичності або неоднорідності в наборі даних. Інформаційний приріст, по суті, показує, наскільки успішно певний атрибут упорядковує дані, зменшуючи їхню ентропію при поділі. Чим вищий цей приріст, тим кращим і ефективнішим вважається поділ даних. [13-15]

C4.5:

Алгоритм C4.5, створений Россом Квінланом, став важливим покращенням у порівнянні з попереднім алгоритмом ID3, оскільки він усуває деякі його недоліки. Однією з головних переваг C4.5 є можливість працювати з неперервними даними — він автоматично знаходить оптимальні порогові значення для їх розбиття на категорії. Крім того, цей алгоритм вміє справлятися з пропущеними значеннями, призначаючи їм ймовірнісні ваги, що робить його більш надійним і гнучким для реального застосування.

Ще однією важливою перевагою алгоритму C4.5 є механізм обрізання дерева (pruning), який застосовується після його побудови. Це допомагає уникнути перенавчання, видаляючи ті гілки дерева, які містять мало корисної інформації або засновані на недостатній кількості прикладів. Крім того, замість звичайного інформаційного приросту, C4.5 використовує коефіцієнт приросту (Gain Ratio), що дозволяє уникнути переваги атрибутів з великою кількістю унікальних значень і робить вибір більш збалансованим.

Завдяки цим покращенням, C4.5 є більш робастним до реальних даних і менш схильним до надмірного навчання, ніж його попередник ID3. Проте, він все ще має свої недоліки, такі як повільна робота для великих наборів даних та побудова дерева за жадібною стратегією, що може не завжди приводити до глобально оптимального рішення. [21,22]

CART (Classification and Regression Trees):

Алгоритм CART (Classification and Regression Trees) був розроблений Лео Брейманом та його колегами, є широко застосовуваним інструментом, що ефективно працює як із класифікаційними, так і з регресійними задачами.

Головна особливість алгоритму CART полягає в тому, що він завжди будує дерево з бінарними розгалуженнями - кожен вузол ділиться рівно на два дочірні. Для задач класифікації вибір найкращого поділу ґрунтується на індексі Джині, який оцінює ймовірність помилки при випадковому визначенні класу елемента на основі розподілу класів у підмножині. Чим менше значення індексу Джині, тим точнішим і чистішим вважається розділення. У разі регресії для вибору розгалужень використовують середньоквадратичну помилку (MSE) або схожі метрики, які вимірюють ступінь розсіяння цільових значень у підмножині.

Ще одна важлива особливість алгоритму CART - це метод обрізання дерева, який називають обрізанням за складністю вартості (Cost-Complexity Pruning). Спочатку алгоритм створює повне дерево, а потім поступово зменшує його розмір, відсікаючи зайві гілки, щоб знайти оптимальний баланс між складністю моделі і її точністю. Такий підхід робить CART гнучким інструментом, здатним ефективно працювати як із класифікаційними, так і з регресійними задачами. Водночас, без правильної обрізки дерево може переобучуватися, що є важливим моментом для врахування під час використання алгоритму. [22]

2.4.3 Оптимізація дерев рішень: техніки обрізання дерева

Обрізання (англ. Pruning) дерева є важливим етапом у формуванні ефективних моделей на базі DT, спрямованим на зниження складності моделі та попередження феномену перенавчання. Існує два основні методологічні підходи до виконання цієї операції. Вибір між цими методами залежить від специфіки поставленої задачі, обсягу доступних даних та наявних обчислювальних ресурсів.

Попереднє обрізання (англ. Pre-pruning):

Попереднє обрізання, або рання зупинка, означає припинення зростання дерева ще до того, як воно повністю сформується. Це відбувається за допомогою різних правил, які визначають, чи варто продовжувати розгалуження. Наприклад, це може бути обмеження на максимальну глибину дерева, мінімальна кількість прикладів у вузлі або поріг покращення, нижче якого подальше поділ вважається недоцільним. Хоча такий підхід дозволяє швидше побудувати модель і зменшити ризик перенавчання, він може призвести до того, що дерево не досягне свого найкращого вигляду, пропускаючи потенційно корисні розгалуження на нижчих рівнях.

Післяобробне обрізання (англ. Post-pruning):

Післяобробне обрізання, або обрізання після побудови, дозволяє дереву спочатку максимально розширитися, досконально підігнавшись під тренувальні дані. Лише після цього здійснюється ретроспективний процес видалення зайвих гілок. Рішення про обрізання приймається на основі аналізу показників на валідаційному наборі даних або із застосуванням статистичних тестів, метою яких є підвищення узагальнюючої здатності моделі. Серед поширених методик виділяють метод зменшення помилок та обрізання за принципом вартості-складності. Цей підхід, як правило, забезпечує побудову більш точних та оптимальних дерев, уникаючи пасток локальних оптимумів. Однак він є більш обчислювально затратним, адже вимагає попереднього формування повного дерева та окремого валідаційного набору даних. [31]

2.4.4 Характерні особливості дерев рішень: переваги та обмеження

Дерева рішень є популярним інструментом у машинному навчанні завдяки своїм численним перевагам. Водночас, вони мають і певні обмеження, які важливо враховувати під час їх використання.

Переваги DT:

- Дерева рішень відрізняються простотою розуміння, оскільки їхня структура демонструє послідовність кроків, які ведуть до прийняття рішення.
- Ці моделі є гнучкими, що дозволяє застосовувати їх у різноманітних задачах. Вони ефективно функціонують як у класифікаційних завданнях, де потрібно віднести об'єкт до певної категорії, так і у регресійних завданнях, де прогнозуються числові значення.
- Підтримка як числових, так і категоріальних змінних. Ці моделі гнучко працюють з обома типами даних без потреби в додатковій попередній обробці, спрощуючи підготовку даних.
- На відміну від багатьох інших алгоритмів машинного навчання, DT не вимагають попередньої нормалізації або масштабування вхідних даних. Це спрощує етап підготовки даних і дозволяє працювати з необробленими значеннями ознак. [20-22]

Недоліки DT:

- Перенавчання (англ. Overfitting) виникає, коли модель дерева рішень надто точно підлаштовується під тренувальні дані, захоплюючи випадкові шуми та дрібні особливості, характерні лише для навчальної вибірки. Через це модель втрачає здатність добре працювати з новими, раніше невідомими даними, що свідчить про її слабку здатність до узагальнення.
- Нестабільність: моделі DT можуть виявитися досить ненадійними. Навіть невеликі зміни у вхідних даних або у складі тренувального набору можуть призвести до суттєвих відмінностей у побудованій структурі дерева та, відповідно, у прогнозах. Ця варіативність знижує передбачуваність та стабільність моделі.
- DT можуть мати схильність віддавати перевагу ознакам з великою кількістю категорій або рівнів. Це означає, що під час побудови моделі вона надто концентрується на таких ознаках, часто недооцінюючи інші,

можливо більш важливі. В результаті точність прогнозів може знизитися, оскільки модель не повністю враховує всі важливі аспекти даних [20]

2.4.5 DecisionTreeClassifier

DecisionTreeClassifier - це клас з бібліотеки scikit-learn, який дозволяє створювати та тренувати DT для задач класифікації. Даний клас реалізує алгоритм CART, який використовує критерій Джині або ентропію для розбиття вузлів.

Основні параметри:

- criterion - функція для оцінювання розбиття: "gini" (за замовчуванням) або "entropy".
- max_depth - максимальна глибина дерева.
- min_samples_split - мінімальна кількість зразків для розбиття вузла.
- random_state - фіксує генератор випадкових чисел для відтворюваності результатів.

Процес навчання та візуалізації:

Побудова прогностичної моделі на основі дерева рішень починається з етапу навчання, який виконується за допомогою методу $\text{fit}(X, y)$. Тут X — це матриця вхідних ознак, де кожен рядок відповідає окремому зразку навчальних даних, а кожен стовпець — певній характеристиці цього зразка. Вектор y містить цільові мітки класів, що відповідають кожному зразку у матриці X .

Під час виконання команди $\text{model.fit}(X, y)$ алгоритм поступово аналізує зв'язки між ознаками та відповідними мітками класів. На основі цього він крок за кроком ділить дані у вузлах, вибираючи найкращі правила розбиття та дотримуючись заданих обмежень моделі. В результаті формується дерево рішень, яке відображає приховані закономірності у наданих даних.

Після завершення навчання корисно візуалізувати структуру дерева за допомогою функції `plot_tree()` з бібліотеки `sklearn.tree`. Цей інструмент дає змогу наочно побачити, як приймаються рішення — від кореневого вузла до кінцевих листових вузлів із прогнозами.

Візуалізація чітко відображає умови розбиття на кожному вузлі (наприклад, умова `feature_name <= threshold_value`), а також кількісні характеристики зразків та їхній розподіл за класами. Листові вузли, що є кінцевими елементами дерева, надають інформацію про остаточний прогноз класу.

Можливість налаштувати візуалізацію за допомогою таких параметрів, як `feature_names`, `class_names`, `filled=True` та `rounded=True`, суттєво підвищує зрозумілість і привабливість отриманого зображення. Такий графік не лише допомагає перевірити коректність моделі та виявити перенавчання, але й є важливим інструментом для наочного представлення результатів аналізу. Це особливо цінно у сферах, де прозорість і обґрунтованість прогнозів мають вирішальне значення. [24, 25]

Висновки

1) У цьому розділі було детально розглянуто теоретичні основи деревоподібних структур даних, що допомогло глибше усвідомити їхню важливу роль у сучасних алгоритмах зберігання, обробки та аналізу інформації. Завдяки гнучкості, ефективності пошуку та можливості наочної інтерпретації, дерева залишаються незамінним інструментом у багатьох галузях, зокрема в телекомунікаціях, де їх використовують для моделювання складних взаємозв'язків між даними.

2) Аналіз підтвердив, що бінарні дерева пошуку (BST) характеризуються високою ефективністю при виконанні операцій пошуку, вставки та видалення, що особливо важливо для створення систем управління великими обсягами даних. Вони зберігають свою цінність навіть у пристроях із обмеженими

ресурсами, таких як сенсорні мережі чи IoT-пристрої. Водночас дерева рішень (DT) вирізняються здатністю до класифікації, прогнозування та виявлення прихованих закономірностей, надаючи чіткі й зрозумілі правила прийняття рішень, що має ключове значення для аналізу якості зв'язку та оптимізації телекомунікаційних мереж.

3) Аналіз алгоритмів ID3, C4.5 та CART показав, що завдяки використанню різних критеріїв розщеплення — таких як ентропія, індекс Джині та зниження дисперсії — дерева рішень можуть ефективно адаптуватися до різних завдань, включаючи як класифікаційні, так і регресійні. Використання операції підрізання дозволяє зменшити перенавчання та підвищити здатність моделей узагальнювати, що є критично важливим для застосування дерев у телекомунікаційних системах. Зокрема, ці моделі успішно використовуються для класифікації якості сигналу, оптимізації ресурсів і виявлення аномалій. Отримані теоретичні знання слугують основою для практичної реалізації дерев рішень із застосуванням сучасних інструментів, зокрема мови Python та бібліотеки Scikit-learn.

РОЗДІЛ 3

ВИКОРИСТАННЯ БІБЛІОТЕКИ SCIKIT-LEARN ДЛЯ ПОБУДОВИ ДЕРЕВ РІШЕНЬ

3.1 Загальні характеристики бібліотеки Scikit-learn

Scikit-learn (або sklearn) - це популярна бібліотека з відкритим кодом для машинного навчання на мові Python. Вона базується на таких широко використовуваних інструментах, як NumPy, SciPy та Matplotlib, що забезпечує високу продуктивність і ефективну роботу з числовими даними. Основна мета sklearn — надати розробникам і дослідникам зручний та єдиний інтерфейс для застосування різноманітних алгоритмів машинного навчання, а також інструменти для підготовки даних, оцінки моделей і підбору оптимальних параметрів.

Основні можливості Scikit-learn включають:

- **Класифікації:** Це завдання полягає у віднесенні об'єктів до однієї з заздалегідь визначених категорій. Sklearn надає такі алгоритми, як метод опорних векторів (SVM), логістична регресія, дерева рішень, випадковий ліс, KNN, Наївний Баєс тощо.
- **Регресії:** основною метою є прогнозування безперервного числового значення на основі вхідних даних. Серед доступних алгоритмів: лінійна регресія, поліноміальна регресія, дерева рішень для регресії, метод опорних векторів для регресії (SVR), градієнтний бустинг.
- **Кластеризації:** представляє собою процес групування схожих об'єктів без попередньо заданих категорій. Реалізовані алгоритми: K-means, DBSCAN, агломеративна кластеризація.
- **Зниження розмірності:** суть у зменшенні кількості змінних (ознак) у наборі даних при збереженні найважливішої інформації. Це допомагає

долати "прокляття розмірності" та візуалізувати дані. Яскраві приклади: Метод головних компонент (PCA), T-SNE.

- Вибір та оцінка моделей: `sklearn` надає потужні інструменти для оцінки ефективності моделей (метрики, такі як точність, повнота, F1-міра, R-квадрат, ROC-крива), а також для вибору найкращих параметрів моделі (Grid Search, Random Search, крос-валідація).
- Попередня обробка даних: функції для масштабування, нормалізації, кодування категоріальних змінних та заповнення пропущених значень. [23, 25]

3.2 Інструменти Scikit-learn для побудови дерева рішень

Для завдань розподілу даних за категоріями, Scikit-learn пропонує зручний і ефективний інструмент - клас `DecisionTreeClassifier`. Він дозволяє будувати моделі дерева рішень, спираючись на добре відомий алгоритм CART (Classification and Regression Tree). Цей клас забезпечує повний набір функцій для навчання, створення прогнозів, візуалізації та всебічного аналізу отриманих моделей. [21]

Ключові етапи побудови дерева рішень за допомогою Scikit-learn:

- Створення моделі:

```
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier()
```
- Навчання моделі (метод `fit`):

```
model.fit(X_train, y_train)
```
- Прогнозування результату (метод `predict`):

```
y_pred = model.predict(X_test)
```
- Візуалізація дерева (метод `plot_tree`):

```
from sklearn.tree import plot_tree
plot_tree(model)
```

Основні параметри:

- criterion: критерій розбиття ("gini" або "entropy")
- max_depth: максимальна глибина дерева
- min_samples_split: мінімальна кількість зразків для поділу вузла
- random_state: для відтворюваності результатів

3.2.1 Порівняльна таблиця: Decision Tree, Logistic Regression, Random Forest

Таблиця 3.1

Порівняння дерева рішень з іншими алгоритмами класифікації

Критерій порівняння	Decision Tree	Логістична регресія	Random Forest / Gradient Boosting
Інтерпретованість	Висока (структура дерева наочна)	Помірна (коефіцієнти моделі)	Низька (сукупність дерев важко інтерпретувати)
Підтримка типів ознак	Числові та категоріальні	Переважно числові (категорії кодують)	Числові та категоріальні
Схильність до перенавчання	Висока	Помірна	Низька
Якість класифікації	Залежить від налаштувань	Висока для лінійно роздільних даних	Висока (завдяки ансамблю моделей)
Виявлення важливих ознак	Так	Обмежено	Так
Обчислювальна складність	Низька–середня	Низька	Висока

Таким чином, із представленого порівняння видно, що DT має перевагу в простоті інтерпретації результатів та універсальності щодо типів вхідних

даних - воно однаково добре працює з числовими та категоріальними ознаками. Завдяки цьому метод часто застосовується для швидкого створення базових моделей та первинного аналізу. Водночас для більш складних або зашумлених наборів даних варто звернути увагу на ансамблеві алгоритми, зокрема Random Forest, які забезпечують вищу стабільність і точність, хоч і потребують більше обчислювальних ресурсів. [20-22]

3.3 Практичне застосування Scikit-learn у телекомунікаціях

Ця бібліотека широко використовується в телекомунікаційній сфері для розв'язання важливих задач. З її допомогою можна прогнозувати відтік абонентів, визначаючи клієнтів, які мають високу ймовірність змінити оператора. Аналіз базується на їхній історії дзвінків, активності в мережі та зверненнях зі скаргами.

Scikit-learn також ефективна у виявленні шахрайства шляхом аналізу нетипових патернів використання послуг.

Для оптимізації якості обслуговування (QoS), бібліотека допомагає аналізувати показники сигналу, швидкості передачі даних та затримок, що дозволяє виявити проблемні ділянки мережі або окремих абонентів.

Крім того, її можна використовувати для аналізу сигналів та шуму, класифікуючи або регресуючи за витягнутими ознаками, що сприяє діагностиці несправностей. Прогнозування завантаження мережі та сегментація клієнтів за їхньою поведінкою є також важливими напрямками застосування, що дозволяють компаніям краще розподіляти ресурси та пропонувати персоналізовані послуги.

Висновки

1) У розділі було досліджено бібліотеку Scikit-learn як один із найбільш зручних та популярних інструментів для побудови дерев рішень. Було доведено важливість використання бібліотеки Scikit-learn у

телекомунікаційних задачах. Завдяки своєму широкому функціоналу бібліотека допомагає розв'язувати прикладні завдання, серед яких - аналіз характеристик сигналу, виявлення шуму та класифікація мережевого трафіку. Ці можливості особливо корисні для розробки інтелектуальних систем моніторингу мереж та оптимізації їхньої роботи.

2) Було розглянуто клас `DecisionTreeClassifier`, що є базовим компонентом для класифікаційних задач. Проаналізовано ключові методи цієї бібліотеки, зокрема `fit()` (навчання моделі), `predict()` (отримання прогнозу) та `plot_tree()` (візуалізація дерева), що дозволяють швидко та ефективно реалізувати моделі у практичних завданнях.

РОЗДІЛ 4

ПОБУДОВА МОДЕЛІ ДЕРЕВА РІШЕНЬ ДЛЯ КЛАСИФІКАЦІЇ ЯКОСТІ ЗВ'ЯЗКУ

Першим кроком для реалізації моделі класифікації якості зв'язку стало встановлення програмного середовища для розробки. У моєму випадку було використано Visual Studio Code, що підтримує інтеграцію з Python, а також має зручний інтерфейс для написання та тестування коду.

Для побудови дерева рішень було використано мову програмування Python разом із бібліотеками scikit-learn (sklearn), matplotlib та graphviz для візуалізації результатів.

Встановлення необхідних пакетів здійснювалося через менеджер пакетів pip. [22]

Після встановлення середовища розробки та бібліотек було створено вхідний набір даних, що складається з трьох параметрів:

- Signal Strength (сила сигналу),
- Noise Level (рівень шуму),
- Packet Loss (втрата пакетів).

Ці характеристики обрано як ключові чинники, що впливають на якість телекомунікаційного з'єднання. Вхідні значення кодувалися як числові категорії - наприклад, сила сигналу: 0 - слабкий, 1 - середній, 2 - сильний.

Також були задані мітки класів якості зв'язку:

- 0 - погана якість,
- 1 - хороша якість.

Нижче наведено код, який демонструє повний процес навчання та візуалізації дерева рішень:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
import matplotlib.pyplot as plt
```

```
# Вхідні дані: [Signal Strength, Noise Level, Packet Loss]
```

```
X = [
```

```

[2, 0, 0], # сильний сигнал, низький шум, без втрат
[1, 0, 0], # середній сигнал, низький шум, без втрат
[0, 1, 1], # слабкий сигнал, високий шум, з втратами
[1, 1, 1], # середній сигнал, високий шум, з втратами
[2, 1, 0], # сильний сигнал, високий шум, без втрат
[0, 0, 1], # слабкий сигнал, низький шум, з втратами
]

# Мітки якості зв'язку: 0 = Погана, 1 = Хороша
y = [1, 1, 0, 0, 1, 0]

# Навчання дерева рішень
clf = DecisionTreeClassifier()
clf = clf.fit(X, y)

# Візуалізація дерева
plt.figure(figsize=(8,5))
tree.plot_tree(
    clf,
    filled=True,
    feature_names=["Signal Strength", "Noise Level", "Packet Loss"],
    class_names=["Poor", "Good"]
)
plt.show()

```

У результаті виконання коду отримано графічне представлення дерева рішень, яке дозволяє відстежити логіку прийняття рішень моделі. За умов слабого сигналу та наявності втрат модель класифікує якість як «погану», тоді як за сильної сили сигналу та відсутності шуму - як «хорошу». На Рисунку 4.2 показано побудовану модель дерева рішень для розпізнавання якісного зв'язку після запуску коду.

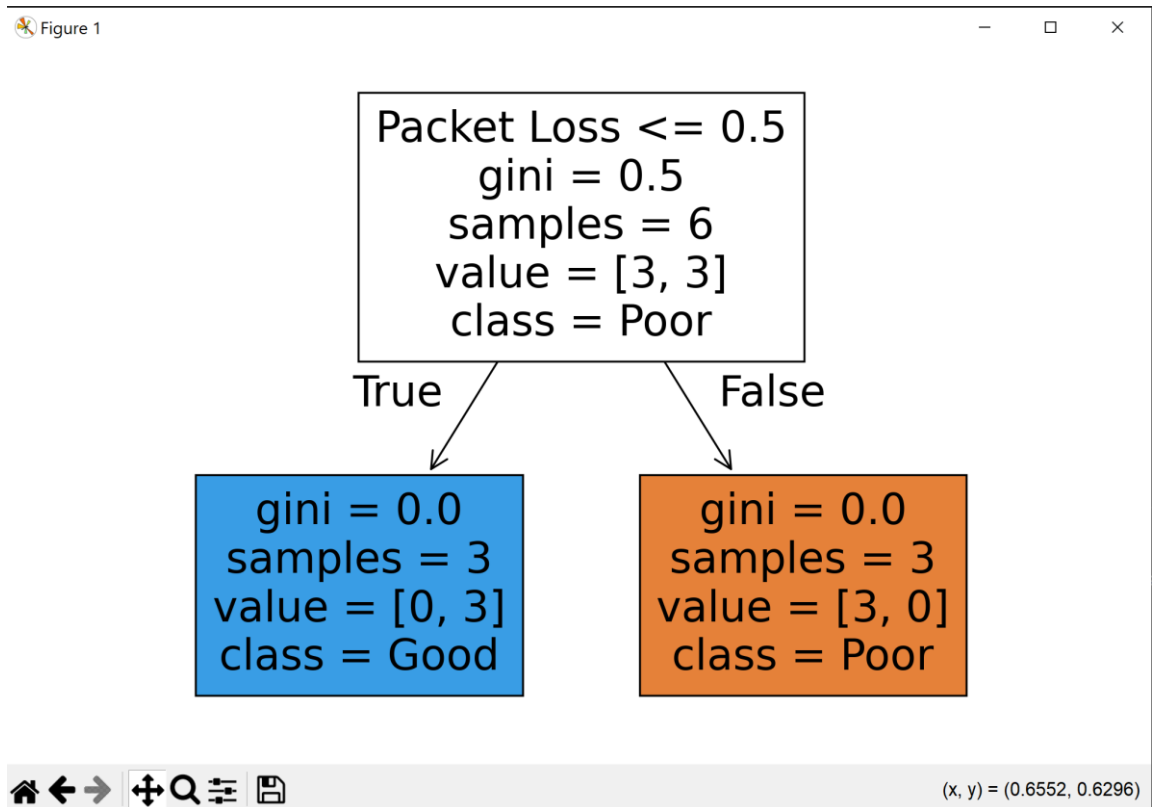


Рис. 4.1 Модель дерева рішень, побудована для розпізнавання якісного та неякісного зв'язку

Висновки

1) У розділі було розглянуто практичний приклад побудови моделі дерева рішень для класифікації якості телекомунікаційного сигналу за допомогою бібліотеки Scikit-learn. Модель була навчена на основі трьох ключових характеристик: сила сигналу, рівень шуму та наявність втрат пакетів, що є типовими параметрами для оцінки якості мережі.

2) Побудована модель дозволяє класифікувати стан зв'язку як «добрий» або «поганий», що має важливе значення для моніторингу та управління мережевими ресурсами. Такий підхід дає змогу своєчасно реагувати на погіршення якості сервісу та вживати заходів для оптимізації роботи мережі.

3) Важливою складовою розглянутого прикладу є візуальне представлення структури дерева рішень, яке дає змогу чітко зрозуміти логіку прийняття рішень. Це особливо цінно в телекомунікаційній практиці, де

інженери повинні мати прозорі й зрозумілі інструменти для аналізу мережесих параметрів і прийняття рішень щодо їхнього вдосконалення.

ЗАГАЛЬНІ ВИСНОВКИ ПО РОБОТІ

Основним завданням моєї дипломної роботи було дослідження деревоподібних структур даних та їх застосування у сфері телекомунікацій, зокрема побудова дерева рішень для оцінки якості сигналу з використанням бібліотеки Scikit-learn. У процесі роботи також було вивчено принципи побудови бінарних дерев пошуку, а також розглянуто алгоритми машинного навчання, які лежать в основі дерева рішень.

У дипломній роботі було досліджено методи зберігання та оброблення даних за допомогою деревоподібних структур. Зокрема, розглянуто бінарні дерева пошуку (BST) та дерева рішень (DT) як базові інструменти ефективної організації, аналізу та інтерпретації даних.

Було проведено теоретичне обґрунтування доцільності використання деревоподібних структур у задачах оброблення інформації, що виникають у сфері телекомунікацій. Показано, що бінарні дерева дозволяють забезпечити швидкий доступ до даних, а дерева рішень - реалізувати моделі прийняття рішень.

Проаналізовано основні властивості та переваги бінарних дерев пошуку. Підкреслено їхню ефективність для реалізації базових операцій (вставка, пошук, видалення) в структурах даних, що використовуються в телекомунікаційних системах.

Розглянуто метод дерева рішень як один із базових алгоритмів машинного навчання. Надано опис його структури, принципів побудови та особливостей застосування в задачах класифікації даних.

У роботі виконано огляд бібліотеки Scikit-learn, яка забезпечує зручні інструменти для реалізації моделей дерева рішень у середовищі Python. Детально розглянуто клас DecisionTreeClassifier, а також ключові методи: `fit()`, `predict()`, `plot_tree()`. Це дозволило реалізувати повноцінну модель класифікації на основі реальних прикладів.

Проведено коротке порівняння дерева рішень з іншими алгоритмами класифікації. Установлено, що дерево рішень має переваги у вигляді простоти інтерпретації та гнучкості до різних типів даних, але поступається ансамблевим методам (Random Forest) у стійкості до перенавчання.

У практичній частині роботи реалізовано модель дерева рішень з використанням Scikit-learn для умовно-телекомунікаційних даних. Модель продемонструвала здатність ефективно аналізувати входні параметри та приймати рішення щодо їх класифікації, що підтверджує можливість її застосування в реальних задачах аналізу сигналу, виявлення шуму чи втрат у каналах зв'язку.

Отримані результати мають практичне значення для впровадження у прикладні задачі інтелектуальної обробки даних, а також можуть бути використані у навчальному процесі при вивченні структур даних, алгоритмів машинного навчання та засобів Python-програмування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. W3Schools. Introduction to Data Structures and Algorithms [Електронний ресурс] – Режим доступу: https://www.w3schools.com/dsa/dsa_intro.php
2. GeeksforGeeks. Data Structures [Електронний ресурс] – Режим доступу: <https://www.geeksforgeeks.org/data-structures/>
3. GeeksforGeeks. Introduction to Linear Data Structures [Електронний ресурс] – Режим доступу: <https://www.geeksforgeeks.org/introduction-to-linear-data-structures/>
4. Unstop. What is Linear Data Structure [Електронний ресурс] – Режим доступу: <https://unstop.com/blog/what-is-linear-data-structure>
5. CCBP Blog. Non-Linear Data Structure [Електронний ресурс] – Режим доступу: <https://www.ccbp.in/blog/articles/non-linear-data-structure>
6. ScholarHat. Non-Linear Data Structures [Електронний ресурс] – Режим доступу: <https://www.scholarhat.com/tutorial/datastructures/non-linear-data-structure>
7. GeeksforGeeks. DSA Tutorial – Learn Data Structures and Algorithms [Електронний ресурс] – Режим доступу: <https://www.geeksforgeeks.org/dsa-tutorial-learn-data-structures-and-algorithms/>
8. GeeksforGeeks. Difference Between Linear and Non-Linear Data Structures [Електронний ресурс] – Режим доступу: <https://www.geeksforgeeks.org/difference-between-linear-and-non-linear-data-structures/>
9. GeeksforGeeks. Introduction to Tree Data Structure [Електронний ресурс] – Режим доступу: <https://www.geeksforgeeks.org/introduction-to-tree-data-structure/>
10. W3Schools. Theory of Trees [Електронний ресурс] – Режим доступу: https://www.w3schools.com/dsa/dsa_theory_trees.php
11. GeeksforGeeks. Binary Tree Data Structure [Електронний ресурс] – Режим доступу: <https://www.geeksforgeeks.org/binary-tree-data-structure/>

- 12.GeeksforGeeks. Types of Trees in Data Structures [Електронний ресурс] – Режим доступу: <https://www.geeksforgeeks.org/types-of-trees-in-data-structures/>
- 13.GeeksforGeeks. Introduction to Binary Search Tree [Електронний ресурс] – Режим доступу: <https://www.geeksforgeeks.org/introduction-to-binary-search-tree/>
- 14.GeeksforGeeks. Insertion in Binary Search Tree [Електронний ресурс] – Режим доступу: <https://www.geeksforgeeks.org/insertion-in-binary-search-tree/>
- 15.GeeksforGeeks. Deletion in Binary Search Tree [Електронний ресурс] – Режим доступу: <https://www.geeksforgeeks.org/deletion-in-binary-search-tree/>
- 16.GeeksforGeeks. Binary Search Tree Set 1: Search and Insertion [Електронний ресурс] – Режим доступу: <https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/>
- 17.GeeksforGeeks. Applications, Advantages and Disadvantages of BST [Електронний ресурс] – Режим доступу: <https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-binary-search-tree/>
- 18.Stanford CS Library. Binary Trees [Електронний ресурс] – Режим доступу: <http://cslibrary.stanford.edu/110/BinaryTrees.pdf>
- 19.Лекції з курсу «Структури даних», Київський університет [Електронний ресурс] – Режим доступу: http://om.univ.kiev.ua/users_upload/15/upload/file/pr_lecture_23.pdf
- 20.GeeksforGeeks. Decision Tree [Електронний ресурс] – Режим доступу: <https://www.geeksforgeeks.org/decision-tree/>
- 21.GeeksforGeeks. Decision Tree Introduction with Example [Електронний ресурс] – Режим доступу: <https://www.geeksforgeeks.org/decision-tree-introduction-example/>
- 22.GeeksforGeeks. Implementing Decision Tree with Scikit-learn [Електронний ресурс] – Режим доступу: <https://www.geeksforgeeks.org/building-and->

[implementing-decision-tree-classifiers-with-scikit-learn-a-comprehensive-guide/](#)

- 23.Scikit-learn Documentation [Електронний ресурс] – Режим доступу: <https://scikit-learn.org/stable/index.html>
- 24.Solver.com. Classification Tree [Електронний ресурс] – Режим доступу: <https://www.solver.com/classification-tree>
- 25.Wikipedia. Scikit-learn [Електронний ресурс] – Режим доступу: <https://en.wikipedia.org/wiki/Scikit-learn>
- 26.Bible P.W., Moser L. *An Open Guide to Data Structures and Algorithms*. – Palni Press, Indianapolis, IN.
- 27.Knuth D.E. *The Art of Computer Programming*. – Addison-Wesley, 1968.
- 28.Левін В.М. *Алгоритми та структури даних*. – Київ: КНУ, 2020.
- 29.Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. *Introduction to Algorithms*. – 3rd ed., MIT Press, 2009.
- 30.*Data Structures and Algorithms: Annotated Reference with Examples*. – First Edition.
- 31.GeeksforGeeks. Pruning Decision Trees [Електронний ресурс] – Режим доступу: <https://www.geeksforgeeks.org/pruning-decision-trees/>
- 32.Mingers, J. & White, L. (2010). A review of the recent contribution of systems thinking to operational research and management science. [Електронний ресурс]. – Режим доступу: <https://core.ac.uk/download/pdf/30284463.pdf>
- 33.Buntine, W. (2009). Theory and Applications of Probabilistic Topic Models. In: Buntine, W., Grobelnik, M., Mladenic, D., Shawe-Taylor, J. (eds) *Machine Learning and Knowledge Discovery in Databases. ECML PKDD 2009. Lecture Notes in Computer Science*, vol 5781. Springer, Berlin, Heidelberg. [Електронний ресурс]. – Режим доступу: <https://doi.org/10.1007/978-3-642-04128-0>
- 34.Bagdonavičius, J. (2012). Sprendimų medžiai ir jų taikymas duomenų klasifikavimui [Електронний ресурс] / J. Bagdonavičius. – Vytauto Didžiojo

Universitetas.

–Режим

доступу:

<https://vb.vdu.lt/object/elaba:2051083/2051083.pdf>

35.Scikit-learn: Decision Trees – [Электронный ресурс]. – Режим доступа:

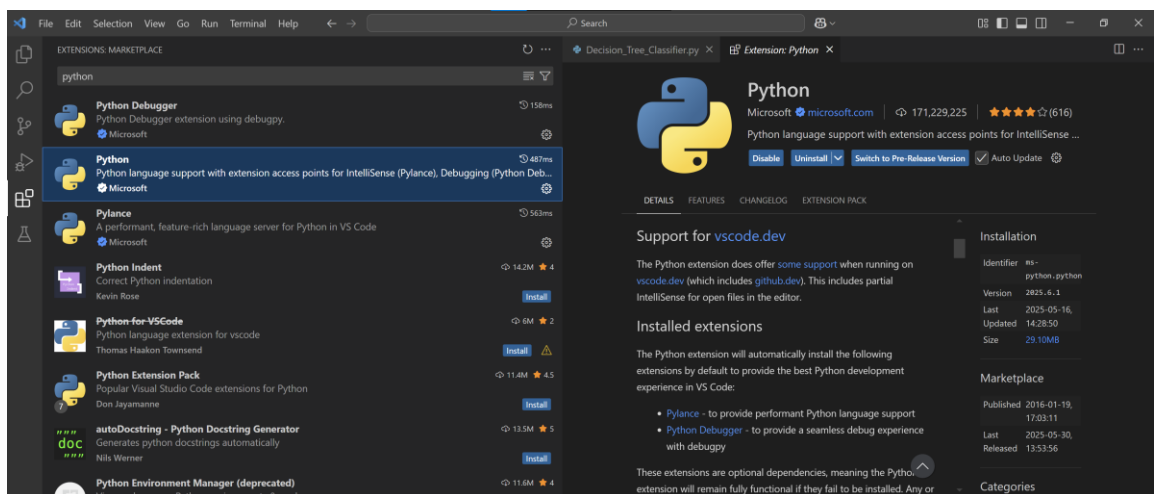
https://code.i-harness.com/it/docs/scikit_learn/modules/tree

ДОДАТОК 1

Налаштування середовища Python у Visual Studio Code

У процесі підготовки до реалізації практичної частини я скористалася середовищем Visual Studio Code. Щоб забезпечити коректну роботу з мовою програмування Python, я встановила відповідне розширення.

- 1) Спочатку я відкрила програму Visual Studio Code.
- 2) Далі, для переходу до розділу розширень, скористалася комбінацією клавіш Ctrl+Shift+X.
- 3) У вікні пошуку розширень ввела “Python”. Серед запропонованих результатів обрала офіційне розширення Python від компанії Microsoft, яке має найбільшу кількість завантажень, і натиснула кнопку “Install”.



ДОДАТОК 2

Інструкція з встановлення необхідних бібліотек Python (scikit-learn, matplotlib).

Перед початком роботи необхідно встановити зазначені бібліотеки за допомогою менеджера пакетів `pip`. Це можна зробити через термінал або командний рядок.

Інструкція з встановлення:

Встановлення `scikit-learn`: `pip install scikit-learn`

Встановлення `matplotlib`: `pip install matplotlib`



```
Командний рядок
Requirement already satisfied: scipy>=1.6.0 in c:\users\oleksandra\appdata\local\programs\python\python313\lib\site-packages (from scikit-learn) (1.15.2)
Requirement already satisfied: joblib>=1.2.0 in c:\users\oleksandra\appdata\local\programs\python\python313\lib\site-packages (from scikit-learn) (1.4.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in c:\users\oleksandra\appdata\local\programs\python\python313\lib\site-packages (from scikit-learn) (3.5.0)
Requirement already satisfied: contourpy>=1.0.1 in c:\users\oleksandra\appdata\local\programs\python\python313\lib\site-packages (from matplotlib) (1.3.2)
Requirement already satisfied: cycler>=0.10 in c:\users\oleksandra\appdata\local\programs\python\python313\lib\site-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in c:\users\oleksandra\appdata\local\programs\python\python313\lib\site-packages (from matplotlib) (4.58.0)
Requirement already satisfied: kiwisolver>=1.3.1 in c:\users\oleksandra\appdata\local\programs\python\python313\lib\site-packages (from matplotlib) (1.4.8)
Requirement already satisfied: packaging>=20.0 in c:\users\oleksandra\appdata\local\programs\python\python313\lib\site-packages (from matplotlib) (25.0)
Requirement already satisfied: pillow>=8 in c:\users\oleksandra\appdata\local\programs\python\python313\lib\site-packages (from matplotlib) (11.2.1)
Requirement already satisfied: pyparsing>=2.3.1 in c:\users\oleksandra\appdata\local\programs\python\python313\lib\site-packages (from matplotlib) (3.2.3)
Requirement already satisfied: python-dateutil>=2.7 in c:\users\oleksandra\appdata\local\programs\python\python313\lib\site-packages (from matplotlib) (2.9.0.post0)
Requirement already satisfied: six>=1.5 in c:\users\oleksandra\appdata\local\programs\python\python313\lib\site-packages (from python-dateutil>=2.7->matplotlib) (1.17.0)

[notice] A new release of pip is available: 24.3.1 -> 25.1.1
[notice] To update, run: python.exe -m pip install --upgrade pip

C:\Users\Oleksandra>
C:\Users\Oleksandra>
```