ЗАКАРПАТСЬКИЙ УГОРСЬКИЙ ІНСТИТУТ ІМЕНІ ФЕРЕНЦА РАКОЦІ ІІ

II. RÁKÓCZI FERENC KÁRPÁTALJAI MAGYAR FŐISKOLA

Кафедра математики та інформатики
Matematika és Informatika Tanszék

АЛГОРОТМИ ТА СТРУКТУРИ ДАНИХ/ ALGORITMUSOK ÉS ADATSZERKEZETEK

МЕТОДИЧНІ ВКАЗІВКИ ДО САМОСТІЙНОЇ РОБОТИ/ MÓDSZERTANI ÚTMUTATÓ AZ ÖNÁLLÓ MmunkÁHOZ

Алгоритми та структури даних/ Algoritmusok és adatszerkezetek
(назва навчальної дисципліни/a tantárgy neve)
Перший (бакалаврський) / Alapképzés (BSc)
(рівень вищої освіти / felsőoktatás szintje)
01 Освіта/Педагогіка / 01 Oktatás/Pedagógia
(галузь знань / képzési ág)
Середня освіта (Інформатика) / Középiskolai oktatás (Informatika)
(освітня програма / képzési program)

Берегове / Beregszász 2025 р.

Методичні вказівки «Алгоритми та структури даних» до самостійної роботи з дисципліни «Алгоритми та структури даних» розроблені на основі Освітньої програми підготовки бакалаврів з галузі знань «01 Освіта/Педагогіка» за напрямом «014 Середня освіта (Інформатика)» як для студентів денної, так і заочної форми навчання. Метою методичного посібника є ознайомлення студентів з основними структурами даних для типових алгоритмах обробки даних, які вивчаються в курсі «Алгоритми та структури даних». У роботі описані основні структури даних, їх властивості, а також типові задачі, для яких вони використовуються. Наведені програми розв'язування задач на мові програмування Python. Методичні вказівки можуть бути використані при самостійній роботі по даному курсу.

Затверджено до використання у навчальному процесі
на засіданні кафедри математики та інформатики ЗУІ ім. Ф.Ракоці ІІ
(протокол № 10 від «23 » червня 2025 року)

Розглянуто та рекомендовано Навчально-методичною радою
Закарпатського угорського інституту імені Ференца Ракоці ІІ
(протокол № 7 від «26 » серпня 2025 року)

Рекомендовано до видання в електронній формі (PDF)
рішенням Вченої ради Закарпатського угорського інституту імені Ференца Ракоці ІІ
(протокол № 8 від «28» серпня 2025 року)

Підготовлено до видання в електронній формі (PDF) кафедрою математики та інформатики спільно з Видавничим відділом Закарпатського угорського інституту імені Ференца Ракоці ІІ

Розробник методичних вказівок:

Йожеф ГОЛОВАЧ – професор кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, доктор технічних наук

Рецензенти:

Федір ГЕЧЕ – професор УжНУ, доктор техніічих наук, професор

Олександр ТИЛИЩАК – професор кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, доктор фізико-математичних наук, доцент

Відповідальні за випуск:

Каталін КУЧІНКА – завідувач кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, доцент, кандидат фізико-математичних наук

Олександр ДОБОШ – начальник Видавничого відділу ЗУІ ім. Ф.Ракоці ІІ

Видавництво: Закарпатський угорський інститут імені Ференца Ракоці ІІ (адреса: пл. Кошута 6, м. Берегове, 90202. Електронна пошта: foiskola@kmf.uz.ua)

Az „Algoritmusok és adatszerkezetek” módszertani utasításokat az „Algoritmusok és adatszerkezetek” tárgy az önálló munkához a BSc „Középiskolai oktatás (Informatika)” képzési programja alapján lett kidolgozva nappali és levelező tagozatos hallgatók részére. A módszertani kézikönyv célja az adatszerkezetek gyakorlati alkalmazása tipikus adatfeldolgozási algoritmusokban, amelyeket a hallagatók az „Algoritmusok és adatstruktúrák” tárgyban tanulják. Az útmutató ismerteti a fontosabb adatstruktúrákat, azok tulajdonságait, valamint azokat a tipikus feladatokat, amelyekhez használják őket. Tartaámazza a programokat Python programozási nyelven, amelyek alkalmazhatók a feladatok megoldására. A módszertani utasítások használhatók a hallgatók önálló munkája során.

Az oktatási folyamatban történő felhasználását jóváhagyta a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Matematika és Informatika Tanszéke

(2025.06.23., 10 számú jegyzőkönyv).

Megjelentetésre javasolta a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola
Oktatási és Módszertani Tanácsa
(2025.08,26., 7. számú jegyzőkönyv).

Elektronikus formában (PDF fájlformátumban) történő kiadásra javasolta
a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Tudományos Tanácsa
(2025.08.28., 8. számú jegyzőkönyv).

Kiadásra előkészítette a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola
Matematika és Informatika Tanszéke, valamint Kiadói Részlege.

A módszertani útmutató kidolgozója:

HOLOVÁCS József – professzor, műszaki tudományok doktora, a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Matematika és Informatika Tanszékének professzora

Szakmai lektorok:

GECSE Ferenc – Az UNE professzora, műszaki tudományok doktora, professzor

TILISHYAK Sandor – a fizikai és matematikai tudományok doktora, docens, a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Matematika és Informatika Tanszékének professzora

A kiadásért felelnek:

KUCSINKA Katalin – PhD, a fizikai és matematikai tudományok kandidátusa, a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Matematika és Informatika Tanszékének tanszékvezetője és docense

DOBOS Sándor – a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Kiadói Részlegének vezetője

A segédlet tartalmáért kizárólag a módszertani útmutató kidolgozója felel.

Kiadó: II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola (cím: 90 202, Beregszász, Kossuth tér 6. E-mail: foiskola@kmf.uz.ua)

ЗМІСТ

ПЕРЕДМОВА

Структури даних використовуються в інформатиці та програмуванні для систематичного зберігання та впорядкування даних. Існують різні типи структур даних, які підходять для зберігання, маніпулювання та доступу до даних різними способами. Структура даних – це контейнер, у якому дані розташовані в певному порядку. Це робить структуру даних ефективною для одних операцій та неефективною для інших. Наша мета – вибрати структури даних, які найбільше підходять для конкретного завдання. ЦІ методичні розробки можуть бути використані для самостійної роботи студентів-інформатиків при вивченні курсу «Алгоритми та структури даних». Спочатку коротко розглянуті типові структури алгоритмів, а потім основна увага приділена вивченню алгоритмізаціх задач для найбільш складних структур даних: дерев та графів. Алгоритми описані на мові Python. Для реалізації програм на Python пропонується використовувати інтегроване середовище розробки PyCharm, створене компанією JetBrains.

1. Алгоритми та структури даних у Python

Алгоритми – це кроки або інструкції, які допомагають вирішити завдання чи проблему. Вони показують, що робити і в порядку, щоб отримати потрібний результат.

Структури даних – це способи впорядкування та зберігання інформації, щоб можна було легко поводитися та працювати з даними. Це, наприклад, як організувати списки або коробки, щоб було зручно знаходити та використовувати те, що в них лежить.

Водночас алгоритми та структури даних дозволяють програмам працювати швидше та ефективніше, тому що вони допомагають розумно обробляти інформацію та знаходити рішення для різних завдань.

Алгоритми Python — це набір інструкцій, написаних мовою програмування Python, які визначають порядок виконання операцій для вирішення певного завдання. Алгоритми можуть бути різними за складністю та ефективністю, і вибір правильного алгоритму може суттєво вплинути на продуктивність програми.

Структури даних у Python – це способи організації та зберігання даних у пам'яті комп'ютера. У Python існує безліч вбудованих структур даних, таких як списки, кортежі, словники та множини, які надають різні способи організації та доступу до даних.

1.1. Програмування алгоритмів на Python

Програмування алгоритмів мовою програмування Python – це написання коду, який реалізує певні алгоритмічні рішення для завдань, включаючи сортування даних, пошук елементів, обчислення математичних функцій та інші операції, які виконуються за певними правилами.

Декілька типових важливих алгоритмів на Python.

1. Сортування списку за допомогою алгоритму бульбашкового сортування

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        for j in range (0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
my_list = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(my_list)
```

```
print("Відсортований список:", my_list)
```

2. Пошук максимального елемента у списку def find_max(arr):

```
def find_max(arr):
    max_value = arr[0]
    for item in arr:
        if item > max_value:
            max_value = item
    return max_value
my_list = [64, 34, 25, 12, 22, 11, 90]
max_element = find_max(my_list)
print("Максимальний елемент:", max_element)
```

3. Обчислення факторіалу числа рекурсивним способом.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial (n - 1)
num = 5
result = factorial(num)
print("Факторіал числа", num, "рівний", result)
```

4. Пошук елемента у списку за допомогою бінарного пошуку

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
            return -1
my_list = [11, 22, 34, 64, 90]
target_value = 34
index = binary_search(my_list, target_value)
print("Індекс елемента", target_value, "у списку:",
index)
```

Python надає багато вбудованих функцій та бібліотек для роботи з даними та реалізації різних алгоритмів.

1.2. Лінійні алгоритми в Python

Лінійні алгоритми – це прості алгоритми, які виконуються послідовно, крок за кроком, без використання розгалужень чи циклів. Вони вирішують завдання, де послідовність виконання операцій є прямолінійною і залежить від зовнішніх умов чи змін даних.

Приклади завдань, які часто вирішуються за допомогою лінійних алгоритмів: обчислення простих математичних операцій, пошук максимального/мінімального елемента у списку, знаходження суми елементів списку та інші прості операції, які виконуються послідовно.

Ось приклади кількох лінійних алгоритмів на Python.

1. Обчислення суми елементів списку

```
def calculate_sum(numbers):
    sum_result = 0
    for num in numbers:
        sum_result += num
    return sum_result
my_list = [1, 2, 3, 4, 5]
result = calculate_sum(my_list)
print("Сума елементів списку:", result)
```

2. Пошук максимального елемента у списку

```
def find_max(numbers):
    max_value = numbers[0]
    for num in numbers:
        if num > max_value:
            max_value = num
    return max_value
my_list = [64, 34, 25, 12, 22, 11, 90]
max_element = find_max(my_list)
print("Максимальний елемент:", max_element)
```

3. Обчислення факторіалу числа

```
def factorial(n):
    if n == 0:
        return 1
    else:
        result = 1
        for i in range(1, n+1):
            result *= i
    return result
num = 5
result = factorial(num)
print("Факторіал числа", num, "рівний", result)
```

Лінійні алгоритми прості та зрозумілі, що робить їх відмінним вибором для вирішення простих завдань, що не потребують складних умов чи циклів.

1.3. Структури даних у Python

У Python існує кілька вбудованих структур даних, які надають різні способи організації та зберігання даних, у тому числі:

1. Списки (Lists)

Списки є одним з найбільш універсальних і часто використовуваних типів структур даних Python. Вони представляють упорядковані колекції елементів, які можуть містити різні типи даних.

Приклад використання списків:

```
# Створення списку
my_list = [1, 2, 3, 4, 5]
```

```
# Додавання елемента до списку
my_list.append(6)
# Зміна значення елемента за індексом
my_list[0] = 10
# Вилучення зрізу списку
slice_list = my_list[1:4]
# Пошук елемента у списку
if 3 in my_list:
    print("Елемент 3 знайдений у списку")
# Видалення елемента зі списку
my_list.remove(2)
print(my_list) # Висновок: [10, 3, 4, 5, 6]
```

2. Кортежі (Tuples)

Кортежі представляють незмінні впорядковані колекції елементів. Вони схожі на списки, але не можуть бути змінені після створення.

Приклад використання кортежів:

```
# Створення кортежу
my_tuple = (1, 2, 3, 4, 5)
# Доступ до елемента кортежу за індексом
print(my_tuple[0]) # Висновок: 1
# Розпакування кортежу
a, b, c, d, e = my_tuple
print(b) # Висновок: 2
```

3. Словники (Dictionaries)

Словники представляють колекції пар ключ-значення, де кожен ключ пов'язаний із певним значенням. Словники забезпечують швидкий доступ до значень ключа.

Приклад використання словників:

```
# Створення словника

my_dict = {"name": "Alice", "age": 30, "city": "New
York"}

# Доступ до значення за ключем

print(my_dict["name"]) # Висновок: "Alice"

# Додавання нової пари ключ-значення

my_dict["occupation"] = "Engineer"

# Перевірка наявності ключа у словнику

if "age" in my_dict:

    print("Ключ 'age' знайдений у словнику")

# Видалення пари ключ-значення зі словника

del my_dict["city"]

print(my_dict) # Висновок: {"name": "Alice", "age": 30,
"occupation": "Engineer"}
```

4. Множини (Sets)

Множини представляють невпорядковані колекції унікальних елементів. Вони надають операції для роботи з об'єднаннями, перетинами та різницями множин.

Приклад використання множин:

```
# Створення множини

my_set = {1, 2, 3, 4, 5}
```

```
# Додавання елемента до множини
my_set.add(6)
# Перевірка наявності елемента у множині
if 3 in my_set:
    print("Елемент 3 знайдений у множині")
# Видалення елемента з множини
my_set.remove(2)
print(my_set) # Результат: {1, 3, 4, 5, 6}
```

Залежно від завдання, вам може знадобитися використовувати той чи інший тип структури даних, щоб ефективно організувати та обробляти дані у програмі.

1.4. Вибір правильної структури даних для завдання

Вибір правильної структури даних для вашого завдання відіграє ключову роль в ефективності та оптимізації програми. Ось покрокова інструкція щодо вибору відповідної структури даних.

1. Визначте основні операції. Перед вибором структури даних, визначте, які операції виконуватимуться. Наприклад, якщо вам потрібно часто виконувати пошук елементів за їх ключами, словник може бути підходящим вибором. Якщо вам необхідно зберігати впорядковані елементи з можливістю повторення, то список (list) або кортеж (tuple) може бути кращим.

2. Враховуйте швидкість доступу. Деякі структури даних забезпечують швидкий доступ до елементів, наприклад, словники та множини. У той час як інші, такі як списки, можуть мати повільний доступ, особливо під час пошуку елементів. Оцініть, наскільки важливим є швидкий доступ до даних у вашому завданні.

3. Враховуйте унікальність елементів. Якщо вам потрібно зберігати унікальні елементи, то множина (set) може бути вибором, оскільки вона автоматично видаляє дублікати. Якщо унікальність не є проблемою, можна використовувати списки або інші структури.

4. Зверніть увагу на операції вставки та видалення. Якщо ваше завдання вимагає частих вставок та видалень елементів, виберіть структуру даних, яка забезпечує ефективні операції додавання та видалення елементів. Списки зазвичай забезпечують швидкі

операції додавання та видалення в кінець, тоді як множини мають ефективні методи додавання та видалення елементів.

5. Розгляньте обсяг даних. Якщо ваше завдання включає великий обсяг даних, то зверніть увагу на пам'ять і швидкість роботи структури даних. Деякі структури можуть споживати більше пам'яті, ніж інші, і забезпечувати повільні операції великого обсягу даних.

6. Врахуйте необхідність сортування. Якщо потрібно підтримувати елементи в упорядкованому вигляді, виберіть структуру даних, яка забезпечує ефективні методи сортування. Списки мають вбудовані методи сортування, тоді як словники та множини зазвичай не гарантують порядок елементів.

7. Враховуйте, що **вибір структури даних може залежати від конкретного контексту та специфіки завдання**. Проводьте тести продуктивності та експериментувати з різними структурами даних, щоб вибрати оптимальне рішення для вашого завдання.

2. Дерева і графи: ієрархічні структури даних

Розглянемо детально алгоритми на деревах та графах, які вважаються складнішими, ніж алгоритми на інших структурах даних. Ці складні ієрархічні структури грають ключову роль структуруванні великих масивів даних, повторюючи складну ієрархію і взаємозв'язки, які у реальних ситуаціях.

Завдяки своїй адаптивності та широкому застосуванню - від складання родоводів до опису організаційних структур, управління файловими системами та вивчення соціальних мереж - ці структури є поширеним і водночас потужним інструментом, що дозволяє організовувати дані та підвищувати зручність пошуку.

У цьому розділі ми поговоримо про складність дерев та графів, розглянемо їх різні форми, характеристики та широкий спектр доступних методів обходу.

Вивчивши ці ієрархічні структури, ви зможете використовувати складні алгоритми, що дасть вам можливість ефективно вирішувати різні завдання в різних областях.

2.1. Дерева: типи та методи обходу

Дерева є типом структури даних, яка повторює структуру ієрархічного дерева. Вони починаються з кореневого значення, яке виступає як початкова точка, і розгалужуються на піддерева, пов'язані з коренем батьківськими вузлами.

Дерева відіграють важливу роль в управлінні базами даних, файлових системах, процедурах прийняття рішень та інших галузях, де ієрархічна організація та подання даних є ключем до впорядкованого та успішного функціонування.

2.1.1. Двійкові дерева

Кожен вузол у цій структурі може мати до двох дочірніх елементів, які зазвичай називають лівим і правим. Двійкові дерева є однією з основних структур даних в інформатиці та знаходять широке застосування у різних додатках.

Вони дозволяють ефективно взаємодіяти з даними, тому активно використовуються у таких операціях, як пошук, сортування та організація інформації. Крім того, двійкові дерева лежать в основі більш складних типів дерев, таких як двійкові дерева пошуку та AVL-дерева, що підвищує їхню корисність і продуктивність.

Двійкові дерева пошуку

Двійкові дерева пошуку (Binary Search Tree, BST) — структура даних, що використовується для представлення даних з ієрархічною організацією, в якій всі вузли відповідають наступній властивості: лівий дочірній вузол менший за батьківський, а правий дочірній — більше. Ця властивість дозволяє виконувати пошук, вставку та видалення за час O(log n), що робить двійкові дерева дуже ефективними.

BST високо цінуються в галузі інформатики та структур даних. Вони пропонують метод ієрархічного зберігання та організації даних, дозволяючи швидко отримувати доступ до них та взаємодіяти з ними.

Дотримання принципів BST допомагає підтримувати баланс та оптимізацію, роблячи роботу зі створення та аналізу алгоритмів ефективною.

Збалансовані дерева

Дерева AVL і червоно-чорні дерева — два популярні приклади деревинного пошуку, що самобалансуються. Вони спеціально розроблені для підтримки балансу шляхом автоматичного коригування своєї структури.

Завдяки цьому регулюванню висота дерева завжди знаходиться під контролем, дозволяючи запобігати зниженню продуктивності та забезпечувати ефективність пошукових операцій. Здатність до самобалансування робить дерева AVL та червоно-чорні дерева надійним та ефективним рішенням для зберігання та пошуку даних.

N-арні дерева

Це тип дерева, де кожен вузол може мати кілька дочірніх. Ця характеристика робить таке дерево менш суворим, ніж двійкове, і дозволяє гнучкіше представляти ієрархічні структури даних.

Такі дерева універсальні і особливо корисні у сценаріях, де дані природним чином утворюють складну ієрархію з безліччю гілок, дозволяючи ефективно організовувати та вилучати дані та керувати ними, а також аналізувати дані в різних галузях, таких як інформатика, біологія та мережеві системи.

B-дерева

B-дерева - структура даних, що використовується в базах даних та файлових системах. Вони дають можливість зберігати величезні обсяги даних та керувати ними. Завдяки своїм унікальним властивостям B-дерева дозволяють ефективно виконувати операції вставки, видалення та пошуку, що робить їх дуже цінним компонентом різних програм.

У базах даних B-дерева відповідають за швидкий доступ та пошук даних, підвищуючи загальну продуктивність. У файлових системах вони сприяють безперешкодній організації файлів та управлінню ними, підвищуючи ефективність операцій.

2.1.2. Методи обходу дерев

Обхід – це процес відвідування всіх вузлів дерева та виконання операції у кожному вузлі. Далі ми розглянемо кілька основних методів обходу та варіанти їх використання.

Обхід по порядку (двійкові дерева)

У методі обходу порядку процес починається з лівого поддерева, рухається до кореня і завершується правим поддеревом. Цей метод часто використовується в двійкових деревах і дозволяє отримувати вузли у відсортованій послідовності, особливо коли йдеться про двійкові дерева пошуку (BST).

При початковому обході лівого піддерева гарантується, що всі вузли дерева будуть відвідані у висхідній послідовності, проходячи через корінь, а потім обхід перейде до вузлів правого піддерева. Така черговість вигідна в кількох сценаріях, наприклад, при пошуку певного ключа в BST або при відображенні дерева в відсортованому вигляді.

Приклад

```
def in_order_traversal(root):
    if root:
        in_order_traversal(root.left)
    print(root.data, end='')
in_order_traversal(root.right)
```

Використання обходу по порядку BST. При цьому методі обходяться вузли BST, починаючи з лівого і закінчуючи самим правим. Такий порядок дозволяє отримати доступ до даних у порядку зростання, що дуже корисно в різних додатках. Яскравим прикладом є створення впорядкованого списку BST, який можна використовувати для різних цілей, таких як аналіз, подальша обробка або надання даних користувачам у доступному форматі.

Обхід у прямому порядку

Почніть з відвідування кореневого вузла дерева, потім досліджуйте ліве піддерево, а потім праве. Цей метод обходу широко використовується для вирішення декількох завдань, таких як дублювання дерева або виконання певних операцій над кожним вузлом. Використання цього методу гарантує, що кожен вузол дерева буде відвіданий та оброблений у потрібній послідовності.

Приклад

```
def pre_order_traversal(root):
    if root:
        print(root.data, end='')
    pre_order_traversal(root.left)
    pre_order_traversal(root.right)
```

Використання обходу у порядку для створення копій дерев. Цей метод передбачає копіювання спочатку кореневого вузла, а потім піддерев'я, гарантуючи, що всі вузли будуть відвідані та скопійовані належним чином, а структура вихідного дерева буде збережена у новій дубльованій копії. Отже, ви можете бути впевнені, що всі ключові елементи дерева зберігатимуться у реплікованій версії. Це особливо корисно, коли порядок вузлів має значення у контексті функціональності чи уявлення дерева.

Крім того, використання обходу у прямому порядку при копіюванні дерева дозволяє змінювати дубльоване дерево. Оскільки структура вихідного дерева зберігається, ви можете легко переміщатися скопійованим і вносити до нього необхідні зміни або доповнення. Це дозволяє легко адаптувати скопійоване дерево до конкретних вимог або змін структури вихідного дерева.

Обхід у зворотному порядку

При цьому методі обходу процес починається з лівого піддерева, переходить до правого і завершується в корені дерева. Метод часто використовується для таких завдань як видалення або звільнення вузлів дерева. Він гарантує, що кожен дочірній вузол буде оброблений до батьківського вузла. Таке впорядковане просування сприяє ефективному керуванню пам'яттю та гарантує ретельну обробку всіх вузлів дерева.

Приклад

```
def post_order_traversal(root):
    if root:
        post_order_traversal(root.left)
        post_order_traversal(root.right)
    print(root.data, end='')
```

Використання обходу у зворотному порядку під час очищення пам'яті. Цей вид обходу визнаний високонадійним та ефективним методом безпечного видалення вузлів у

деревоподібних структурах, особливо у мовах програмування, де управління пам'яттю здійснюється вручну.

Цей метод гарантує, що вузол буде видалено лише після того, як всі його дочірні вузли будуть оброблені належним чином, зберігаючи цілісність та стабільність використання пам'яті у дереві. Застосовуючи цю стратегію, програмісти можуть ефективно управляти пам'яттю та звільняти ресурси, тим самим запобігаючи витоку пам'яті та підвищуючи загальну продуктивність системи.

Обхід за рівнями (завширшки)

При такому обході вузли відвідуються за одним рівнем за раз, починаючи з кореня. Цей метод цінний, коли важливий рівень ієрархії, оскільки гарантує, що це вузли цьому рівні будуть вивчені, як можна переходити до наступного.

Обхід за рівнями дозволяє ретельно досліджувати все дерево, точно відображаючи ієрархічну природу даних, і часто застосовується в таких ситуаціях як вивчення організаційної структури, зображення файлових систем або управління топологіями мереж.

Приклад

```
from collections import deque

def level_order_traversal(root):
    if root is None:
        return
    queue = deque([root])
    while queue:
        node = queue.popleft()
    print(node.data, end='')
    if node.left:
        queue.append(node.left)
    if node.right:
        queue.append(node.right)
```

Використання обходу за рівнями виконання операцій кожному рівні ієрархії. Цей метод особливо вигідний у ситуаціях, що вимагають виконання насамперед, наприклад, при застосуванні алгоритмів пошуку в деревоподібних структурах, тому що в ньому використовується черга.

Завдяки використанню обходу рівнями розробники можуть виконувати операції систематизованим і впорядкованим чином, тим самим підвищуючи ефективність своїх алгоритмів.

Таким чином, кожен метод обходу виконує свою роль, і вибір залежить від конкретних вимог.

Розширені концепції обходу

Обхід Морріса підвищення ефективності використання простору

Обхід Морріса - це метод обходу дерев, який дозволяє відмовитися від використання рекурсії або стека, що призводить до складності O(1). Це означає, що при переміщенні по дереву пам'ять використовуватиметься як мінімум. Замість того, щоб покладатися на додаткові структури даних, обхід Морріса використовує власну структуру дерева для зберігання та доступу до інформації, що підвищує ефективність використання пам'яті.

Незважаючи на складність, цей метод дуже корисний в умовах обмеженого обсягу пам'яті. Використовуючи обхід Морріса, розробники можуть налаштовувати свої алгоритми середовищ з обмеженим обсягом пам'яті, забезпечуючи безперебійну роботу навіть за обмежених ресурсах.

Потокові двійкові дерева для ефективного обходу

Потокове двійкове дерево - варіант двійкового дерева, в якому вводяться додаткові покажчики, щоб оптимізувати процес обходу. У дереві цього звичайні нульові покажчики замінюються покажчиками на порядкового попередника чи наступника.

Таким чином, деревоподібна структура стає взаємопов'язаною і дозволяє швидше і ефективніше виконувати обхід по порядку. Введення цих додаткових покажчиків підвищує здатність дерева до систематичного переміщення його елементами, що сприяє ефективному пошуку даних та взаємодії з ними.

2.1.3. Практичне застосування дерев

Синтаксичні дерева у компіляторах

В інформатиці компілятори є невід'ємною частиною перетворення мов програмування високого рівня на низькорівневий машинний код. Для цього вони часто

використовують деревоподібні структури даних, які виступають ефективним та надійним засобом відображення та аналізу складної структури програми.

Ці дерева дають компіляторам можливість точно та акуратно переміщатися за вихідним кодом та змінювати його. Ця процедура не тільки гарантує точність перекладу, а й дозволяє застосовувати численні оптимізації, тим самим покращуючи виконання програм та підвищуючи продуктивність додатків.

Дерева рішень у машинному навчанні

Дерева рішень - важливий елемент алгоритмів машинного навчання (МО), який допомагає приймати рішення шляхом вивчення закономірностей та зв'язків у вихідних даних. Моделі МО, проходячи через ці дерева та оцінюючи різні гілки та вузли, здатні робити точні прогнози та виконувати класифікації.

Можливість переміщатися по деревах рішень дозволяє алгоритмам машинного навчання отримувати інформацію та формулювати обґрунтовані рішення на основі вихідних даних, що значно підвищує продуктивність та ефективність самої системи МО.

Об'єктні моделі документів у браузерах

Об'єктні моделі документів (Document Object Model, DOM) — ключовий аспект веб-розробки, що впливає на те, як браузери читають веб-сторінки та взаємодіють із ними. По суті, це деревоподібна структура, що представляє різні елементи, що становлять веб-сторінку.

Ця структура дозволяє браузерам не тільки розуміти вміст сторінки, але й змінювати його за необхідності. Переміщаючись по дереву DOM, браузери можуть отримувати доступ до різних елементів, таких як абзаци, заголовки, зображення та посилання, та змінювати їх.

Завдяки цьому браузери можуть відображати веб-сторінки у візуально привабливому вигляді, даючи користувачам можливість отримати досвід інтерактивної та динамічної взаємодії. Таким чином, глибоке розуміння DOM та принципів їх роботи дозволяє розробникам створювати та вдосконалювати сайти, які задовольняють зростаючі потреби та очікування користувачів.

Файлові системи в операційних системах

Файлові каталоги і структури зазвичай представляються як дерев, що дозволяє виконувати ефективний пошук і організацію файлів з допомогою операцій обходу. Файлові системи, що забезпечують ієрархічну структуру, полегшують ефективне зберігання та пошук файлів.

Ці системи також враховують метадані, такі як дозволи на доступ до файлів та тимчасові мітки, що допомагає керувати файлами. Вони підтримують різні типи файлів, такі як документи, зображення та відео, забезпечуючи універсальний підхід до зберігання різних типів даних та доступу до них.

Крім того, файлові системи надають можливості стиснення та шифрування файлів, оптимізуючи простір зберігання та підвищуючи безпеку даних.

Вивчення деревоподібних структур та методів їхнього обходу — не просто навчальне завдання; це практична навичка, що має широке застосування. Поглиблення на цю тему дозволяє побачити, як за допомогою дерев можна вирішувати складні завдання, використовуючи ефективні та елегантні підходи.

Набуваючи досвіду в обході дерев, ви отримуєте надійний інструментарій для вирішення складних завдань та стимулювання інноваційних розробок у галузі інформатики та технологій.

3. Графи: представлення та основні алгоритми

Графи, як структури даних, що легко адаптуються, чудово підходять для уявлення складних відносин між об'єктами. Ми розглянемо структуру графів та деякі важливі алгоритми, розроблені для них.

Ми також обговоримо різні типи графів, у тому числі орієнтовані та неорієнтовані, зважені та незважені, циклічні та ациклічні. Розуміти нюанси цих типів дуже важливо, оскільки завдяки їхнім відмінним властивостям вони застосовуються у моделюванні різних реальних ситуацій.

Крім того, ми розглянемо деякі базові поняття теорії графів: вершини, ребра та шляхи. Вивчення цих елементів дозволяє глибше зрозуміти структуру та поведінку графів.

Графи використовуються в різних областях: від побудови найкоротших маршрутів у мережах до розуміння соціальних зв'язків, аналізу комп'ютерних мереж і навіть моделювання поширення хвороб. Їх здатність представляти та вирішувати різні складні завдання робить їх найважливішим інструментом у таких галузях, як інформатика, математика та соціальні науки.

Основні поняття

Графи - найважливіші структури даних в інформатиці, що складаються з набору вузлів (або вершин) і ребер, що їх пов'язують. Вузли символізують сутності чи елементи, а ребра позначають зв'язок між ними.

Ці структури знаходять широке застосування у різних галузях, як-от соціальні та комп'ютерні мережі, і навіть транспортні системи. Вони дозволяють моделювати та вивчати складні взаємозв'язки та залежності між різними об'єктами. Завдяки візуалізації вузлів та ребер графи допомагають зрозуміти структуру та закономірності, що лежать в основі даних.

Вузли та ребра

Вузли (також звані вершинами) є основними компонентами, що представляють широкий спектр сутностей: об'єктів, людей або будь-яких інших елементів.

Ребра є мостами, що встановлюють зв'язки між цими сутностями. Ці зв'язки є засобом відображення взаємодії різних сутностей або їх залежностей.

Використовуючи вузли та ребра, граф може ефективно відобразити складну взаємодію та динаміку, які існують у даній системі чи мережі.

Орієнтовані та неорієнтовані графи

Орієнтовані графи — тип графів, у яких кожне ребро має певний напрямок, що свідчить про односторонню зв'язок між вузлами. Це означає, що інформація протікає у певному напрямку від одного вузла до іншого.

Неорієнтовані графи - тип графів, що допускає двонаправлені ребра, тобто зв'язки між вузлами можуть бути двосторонніми. Це дозволяє виявити різні закономірності та зв'язки, які можуть бути не відразу очевидними в орієнтованому графі.

Зважені графи

Зважені графи розширюють стандартну концепцію, запроваджуючи додатковий шар, який надає аналізу складності та глибини. Це досягається шляхом надання ребрам числових значень або ваг, які допомагають кількісно оцінити різні елементи, що впливають на зв'язки між вузлами.

Ці ваги можуть символізувати безліч важливих аспектів, таких як вартість, відстань чи інші важливі показники, які хочемо вивчити. Візьмемо, наприклад, транспортну мережу; тут ваги можуть означати відстань між містами, допомагаючи визначити найкоротший чи найефективніший шлях. У соціальних мережах ваги можуть вказувати силу зв'язків між людьми, допомагаючи визначити ключові чинники впливу чи кластери спільноти.

Додавання ваги дозволяє глибше проникнути в структуру графа, виявити закономірності і зрозуміти те, що при аналізі стандартного графа можна упустити. Такий підхід дозволяє більш точно зрозуміти зв'язки між вузлами, що призводить до прийняття більш обґрунтованих рішень та чіткіших висновків.

Представлення графів

Існують два основні способи представлення графів в інформатиці - *матриця суміжності та список суміжності.*

Матриця суміжності

Матриця суміжності - найважливіша структура даних, яка служить зображення графів. Вона задається у вигляді двовимірного масиву, рядки та стовпці якого відповідають вузлам графа. У цьому масиві кожен осередок [i][j] може містити або логічне значення, що показує наявність або відсутність ребра між вузлом і і вузлом j, або числове значення, що визначає вагу ребра, якщо граф є зваженим.

Ця структура дуже ефективна для щільних графів, у яких кількість ребер велика порівняно з максимально можливим. Вона дозволяє швидко та ефективно працювати з даними про зв'язки графа. За допомогою матриці суміжності легко визначити, чи є ребро між будь-якими двома вершинами, і просто отримати вагу ребра у завислих графах.

Приклад

```
class Graph:

def __init__(self, size):

self.adj_matrix = [[0 for _ in range(size)] for _ in
range(size)]

def add_edge(self, start, end):

self.adj_matrix[start][end] = 1

self.adj_matrix[end][start] = 1 # Для неорієнтованого
графа

def remove_edge(self, start, end):

self.adj_matrix[start][end] = 0

self.adj_matrix[end][start] = 0 # Для неорієнтованого
графа


# Приклад використання
```

```
graph = Graph(3)
graph.add_edge(0, 1)
graph.add_edge(1, 2)
print(graph.adj_matrix)
```

Список суміжності

Список суміжності - структура даних, призначена для представлення графів. У ній кожен вузол графа представлений елементом, що містить список вузлів, які є суміжними або безпосередньо з ним.

Основною перевагою списку суміжності є економія місця, особливо в розріджених графах. Це графи з відносно невеликою кількістю ребер, порівняно з максимально можливим. У таких сценаріях списки суміжності вигідні, оскільки вимагають зберігання лише тих вершин, які справді пов'язані між собою, що дозволяє заощаджувати пам'ять.

Використання списку суміжності підвищує ефективність різних операцій, як-от визначення всіх сусідів конкретного вузла чи перевірка зв'язок між двома вузлами. Тому такий список часто використовується в багатьох алгоритмах та додатках, заснованих на графах.

Приклад

```
class Graph:
def __init__(self, size):
self.adj_list = [[] for _ in range(size)]


def add_edge(self, start, end):
self.adj_list[start].append(end)
self.adj_list[end].append(start) # Для неорієнтованого графа


# Приклад використання
graph = Graph(3)
graph.add_edge(0, 1)
graph.add_edge(1, 2)
print(graph.adj_list)
```

Основні алгоритми побудови графів

Пошук у глибину

Пошук у глибину (Depth-First Search, DFS) — алгоритм обходу, який починає з вибраного вузла і проходить по гілці якнайдалі, перш ніж повернутися назад. Цей метод особливо корисний у таких завданнях, як вирішення головоломок, де для того, щоб знайти рішення, потрібно спочатку вивчити всі можливі варіанти.

Використовуючи DFS, ви можете охопити всю область пошуку, методично просуваючись по кожному потенційному шляху. Алгоритм здатний переміщатися великими і заплутаними просторами пошуку, концентруючись однією галузі за раз.

DFS гарантує, що жодне рішення не буде втрачено, оскільки ретельно вивчає кожен шлях обходу, стартуючи з початкової точки. Він пропонує стратегію пошуку, що дозволяє детально вивчити всі можливі шляхи у просторі пошуку.

Приклад

```
def dfs(graph, start, visited=None):
if visited is None:
```

```
visited = set()
visited.add(start)
for neighbor в graph.adj_list[start]:
if neighbor not in visited:
dfs(graph, neighbor, visited)
return visited
```

Пошук у ширину

Пошук у ширину (Breadth-First Search, BFS) - алгоритм обходу, який відвідує вузли порівнево, починаючи з кореневого і рухаючись вшир. Тобто спочатку він перевіряє всі вузли, які є найближчими до кореня, і лише потім переходить на наступний рівень. Цей метод особливо ефективний визначення найкоротшого шляху в незважених графах. Ключова перевага BFS — його здатність гарантувати виявлення найкоротшого шляху між двома вузлами, якщо існує.

Алгоритм BFS починає роботу з обраного вузла: досліджує всіх його найближчих сусідів, а потім переходить до сусідів цих сусідів і т. д. Завдяки цьому BFS гарантує відвідування всіх вузлів графа і знаходження найкоротшого шляху. Таким чином, BFS підходить для додатків, які потребують вирішення проблеми найкоротшого шляху, таких як навігаційні системи або алгоритми мережної маршрутизації.

Приклад

```
from collections import deque
def bfs(graph, start):
visited = set()
queue = deque([start])
while queue:
vertex = queue.popleft()
if vertex not in visited:
visited.add(vertex)
queue.extend(set(graph.adj_list[vertex]) - visited)
return visited
```

Алгоритм Дейкстри (для зважених графів)

Алгоритм Дейкстри – ключовий інструмент для пошуку найкоротших шляхів у графах, особливо корисний, коли ребра графа зважені. Цей алгоритм визначає оптимальний маршрут між двома вузлами з урахуванням ваг ребер та активно використовується в таких областях, як протоколи мережної маршрутизації та системи GPS-навігації.

Важливість алгоритму Дейкстри та її впливом геть різні технологічні системи незаперечні. Розуміючи його основи, ви зможете глибше вивчити теорію графів та варіанти її реального застосування. Ми поговоримо про нього докладніше у наступних розділах.

Таким чином, графи – дуже важлива концепція в інформатиці, що втілює природу реляційних даних та дозволяє аналізувати складні відносини між сутностями.

Вивчення теорії графів не лише збагатить ваше уявлення про графи, а й відкриє спектр алгоритмів та методів для вилучення інформації, вирішення складних завдань та вдосконалення процесів у даних, структурованих графами.

Оволодівши навичками представлення та роботи з графами, ви зможете вирішувати реальні завдання та отримувати важливі результати.

Практичне застосування графів

Соціальні мережі. Графові структури використовують на платформах деяких соціальних мереж. Користувачі представлені у вигляді вузлів, а зв'язки, такі як дружба чи професійна взаємодія, — ребер. Таке графічне уявлення як спрощує візуальну складність мереж, а й допомагає користувачам легко орієнтуватися у великій мережі зв'язків цих платформах.

Маршрутизація в Інтернеті. Маршрутизатори використовують графові алгоритми, у тому числі алгоритм Дейкстри, для пошуку найбільш ефективних шляхів передачі пакетів даних через мережу. Ці алгоритми враховують різні аспекти топології мережі, такі як пропускна здатність каналу, затримка та перевантаженість, щоб виконувати ефективну маршрутизацію пакетів. Такий оптимальний пошук шляхів забезпечує своєчасну доставку даних, скорочує затримки та підвищує загальну ефективність мережі.

Рекомендаційні системи. Платформи електронного продажу та контенту, такі як Amazon і Netflix, використовують алгоритми на основі графів у своїх рекомендаційних системах. Вони пов'язують користувачів з товарами або контентом, що відповідають їх перевагам, на основі індивідуального вибору користувачів. Ці алгоритми аналізують великі дані про користувачів, виявляють тенденції та дають відповідні рекомендації, постійно пропонуючи нові варіанти.

Карти Google. Графічні алгоритми використовуються визначення найбільш ефективних маршрутів між населеними пунктами, враховуючи відстань, трафік, перекриття доріг та інші важливі дані. Завдяки цим удосконаленим алгоритмам карти

Google надають точні дані в режимі реального часу, гарантуючи користувачам безпроблемну подорож.

Практичні поради

У завданнях, пов'язаних з графами, ключове значення має розуміння їхньої природи та вимог. Воно допомагає зробити вибір між матрицею суміжності та списком суміжності. Якщо потрібно швидко перевірити наявність прямого зв'язку між двома вузлами, потрібно використовувати матрицю. І навпаки, списки суміжності є кращими для розріджених графів, де велику роль відіграє економія місця.

Важливо розуміти і те, чи є граф орієнтованим чи неорієнтованим, оскільки це впливає на методи додавання ребер та обходу та дозволяє виконувати точні та ефективні операції.

Крім того, при роботі із зваженими графами, особливо в задачах найкоротшого шляху, дуже важливо враховувати ваги ребер, у тому числі можливість негативних ваг. Оцінка ваги дозволяє вибрати оптимальний алгоритм, щоб отримати точний результат для конкретного завдання.

Таким чином, вивчати теорію графів, їх різні типи та пов'язані з ними алгоритми дуже корисно для розвитку навичок розв'язання задач. .

Практичні вправи

1. Реалізація дерева двійкового пошуку.

Створіть базове двійкове пошукове дерево з методами вставки та обходу по порядку.

Вставте числа 10, 5, 15, 2, 5, 13, 22, 1, 14, а потім виконайте обхід по порядку.

Рішення

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
class BinarySearchTree:
    def __init__(self):
        self.root = None


def insert(self, value):
```

```
        new_node = Node(value)
        if self.root is None:
            self.root = new_node
        return
        current = self.root
        while True:
            if value < current.value:
                if current.left is None:
                    current.left = new_node
                return
                current = current.left
                else:
                    if current.right is None:
                        current.right = new_node
                return
                current = current.right

    def in_order_traversal(self, node, result=[]):
        if node:
            self.in_order_traversal(node.left, result)
            result.append(node.value)
            self.in_order_traversal(node.right, result)
        return result

    # Тест
    bst = BinarySearchTree()
    for value in [10, 5, 15, 2, 5, 13, 22, 1, 14]:
```

```
bst.insert(value)

print(bst.in_order_traversal(bst.root)) # Висновок: [1, 2, 5, 5, 10, 13, 14, 15, 22]
```

2. Реалізація графа з допомогою списку суміжності.

Створіть клас Graph з методами для додавання ребер та пошуку в ширину.

Додайте ребра, що з'єднують вузли 0, 1, 2, 3 і 4 нелінійним чином, і виконайте пошук завширшки, починаючи з вузла 0.

Рішення

```
from collections import defaultdict, deque


class Graph:

def __init__(self):

self.graph = defaultdict(list)


def add_edge(self, u, v):

self.graph[u].append(v)


def bfs(self, start):

visited = set()

queue = deque([start])

while queue:

vertex = queue.popleft()

if vertex not in visited:

print(vertex, end='')

visited.add(vertex)

queue.extend(set(self.graph[vertex]) - visited)
```

```
# Тест
g = Graph()
edges = [(0, 1), (1, 2), (1, 3), (2, 4)]
for u, v in edges:
g.add_edge(u, v)
g.bfs(0) # Висновок: 0 1 2 3 4
```

3. Пошук у глибину на графі.

Реалізуйте пошук у глибину графа, створеного вправі 2.

Виконайте пошук у глибину, починаючи з вузла 0.

Рішення

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end='')
    for neighbor в graph.graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

# Тест
dfs(g, 0) # Вихід: 0 1 2 4 3
```

Висновок

У цьому розділі ми говорили про дерева і графи — дві основні структури даних в інформатиці, про їхню взаємопов'язану природу і про те, як їх можна використовувати для моделювання складних відносин і вирішення різноманітних завдань.

Ми почали з дерев і досліджували їхню ієрархічну природу, завдяки якій вони відмінно підходять для представлення даних, пов'язаних ставленням «предок — нащадок». Ми розглянули прості двійкові дерева, кожен вузол яких має не більше двох

нащадків, і більш складні структури, такі як двійкові пошукові дерева, дерева AVL і B-дерева, а також познайомили вас з варіантами їх застосування.

Далі ми поринули у тему методів обходу дерев. Кожен їх служить певної мети: від сортування даних (обхід по порядку в двійкових деревах пошуку) до копіювання дерев (обхід у порядку) і звільнення місця у пам'яті (обхід у порядку). Крім того, ми обговорили метод обходу рівнями. Всі ці методи не тільки дозволяють отримати доступ до кожного елемента дерева, але і дають більш глибоке уявлення про його структуру та властивості.

Потім ми поговорили про графи — узагальненішу структуру даних, яка може представляти складні відносини між елементами. Ми обговорили, як графи можуть бути представлені за допомогою матриць та списків суміжності, які мають переваги в залежності від конкретної ситуації.

Алгоритми роботи з графами дозволили вам побачити варіанти їх використання у реальних сценаріях, таких як соціальні мережі та інтернет-маршрутизація. Як основні методи обходу чи пошуку у графі були представлені алгоритми пошуку в глибину та ширину. Ми також торкнулися алгоритму Дейкстри для пошуку найкоротшого шляху у зважених графах.

Вправи даного розділу допомогли вам на практиці закріпити отримані знання та покращити навички програмування.

Таким чином, дерева та графи — не просто теоретичні поняття. Ці структури відіграють ключову роль в організації, обробці та отриманні даних і знаходять застосування в різних областях — від технологій до науки і не тільки.

4. Інтегроване середовище розробки PyCharm

PyCharm це кросплатформове, потужне і розумне інтегроване середовище розробки. Цей інструмент надає можливість комфортної роботи з мовою Python, починаючи від автодоповнення коду до потужного відладчика.

PyCharm був розроблений компанією JetBrains, що спеціалізується на розробці інструментів для програмування різними мовами, таких як Java, Kotlin, C#, F#, C++, Ruby, Python, PHP, JavaScript та багато інших. PyCharm має дві версії – Community Edition та Professional Edition. PyCharm Community Edition безкоштовна, а PyCharm Professional Edition платна ліцензія. Обидві версії PyCharm забезпечують повну підтримку мови Python та дозволяють зручно працювати з проектами, проте Professional Edition пропонує розширений набір функцій та інструментів, роблячи її кращим вибором для професійних розробників та комерційних проектів.

Для того, щоб почати використовувати PyCharm, слід:

• встановити PyCharm із офіційного сайту JetBrains, вибравши Community або Professional Edition;

• створити новий проект, вибравши тип (зазвичай Python) та інтерпретатор Python;

• відкрити файли з кодом Python, перетягнувши їх у вікно або через меню "File" -> "Open";

• ознайомитися з інтерфейсом, включаючи редактор коду, вікно проекту та «Tool Windows»;

• ввести код, використовуючи автодоповнення (використовуйте відладчик у разі потреби);

• вивчити основні функції PyCharm, включаючи рефакторинг та роботу з Git;

• розширити можливості PyCharm за допомогою плагінів із «Plugin Repository».

Основні функції PyCharm

PyCharm – це не просто текстовий редактор для програмування на Python, це справжній інтелектуальний помічник, який робить процес програмування зручним та продуктивним.

PyCharm надає розробникам багато ключових функцій, які значно спрощують процес програмування. За допомогою цих функцій PyCharm спрощує процес програмування, допомагає створювати чистіший та ефективніший код, а також підвищує продуктивність розробника.

Ось кілька із них:

• **Автодоповнення та інтелектуальні підказки**. Це дозволяє швидше писати код, не витрачаючи час на пошук імен функцій, методів та змінних, а також скорочує ймовірність помилок під час їхнього виклику.

• **Інспектування коду.** Статичний аналіз коду допомагає виявляти потенційні помилки та підтримувати високу якість коду.

• **Налагодження та профільування.** Можна встановлювати точки зупинки, аналізувати значення змінних і крок за кроком виконувати код, що допомагає зрозуміти, що відбувається в програмі на кожному етапі виконання. Інструменти профілювання дозволяють оптимізувати продуктивність вашого коду.

• **Віртуальне оточення PyCharm.** Це ізольований простір для програм Python, який дозволяє мати свій власний набір залежностей, не впливаючи на інші проекти. Це дозволяє розробляти та тестувати програми в окремих, чистих середовищах, забезпечуючи надійність та запобігаючи конфліктам між залежностями різних проектів.

• **Підтримка фреймворків та технологій,** таких як Django, Flask, SQLAlchemy, HTML, CSS, JavaScript та інші. Це дозволяє розробляти веб-програми та інші проекти з використанням сучасних технологій.

• **Рефакторинг.** Інструменти рефакторингу допомагають змінювати структуру коду без втрати функціональності.

Пояснимо поняття рефакорингу більш детально.

Рефакторинг Python - це процес покращення внутрішньої структури коду Python без зміни його зовнішньої поведінки (функціональності). Мета рефакторинга - зробити код більш зрозумілим, підтримуваним та гнучким для подальших змін.

Рефакторинг:

- включає в себе переписування або переструктурування коду, щоб зробити його більш читабельним, модульним та ефективним.
- Потрібен для покращення читабельності: Чіткий та структурований код легше розуміти та підтримувати.
- Полегшення внесення змін: Зміни у рефакторованому коді більш передбачувані та менш схильні до помилок.
- Підвищення продуктивності: Рефакторинг може покращити швидкість виконання програми, хоча це не є його основною метою.
- Зменшення технічного боргу: Регулярний рефакторинг допомагає уникнути накопичення "технічного боргу" - складного та заплутаного коду, який важко підтримувати.
- Рефакторинг часто починається з виявлення ознак проблем у коді (наприклад, дублювання коду, великі функції, складні умови).

Наприклад*:*

Вилучення методу: Розбиття великої функції на менші, більш конкретні методи.

Вилучення класу: Виділення частини функціональності у окремий клас.

Заміна умовного оператора тернарним виразом: Спрощення умовних виразів.

Використання генераторів списків замість циклів: Більш компактний та виразний спосіб створення списків.

Застосування функцій enumerate та zip: Зручніші способи перебору елементів списків та зв'язування декількох списків.

- **Тестування:** Кожен етап рефакторингу повинен супроводжуватися тестуванням, щоб переконатися, що функціональність програми не змінилася.

Створення проекту в PyCharm.

Для цього потрібно лише кілька кроків:

• запустити PyCharm і вибрати опцію Create New Project на стартовому екрані;

• вибрати тип проекту, який відповідає вашій мові програмування (наприклад, Python, Java, JavaScript тощо);

• вказати папку, де розташовуватиметься ваш проект на комп'ютері;

• вказати інтерпретатор мови програмування, який використовуватиметься у проекті (наприклад, Python, якщо ви створюєте проект Python).

Після натискання кнопки Create ваш новий проект буде створено.

Плагіни для PyCharm

Плагіни для PyCharm – це додаткові розширення, які дозволяють додати нову функціональність у середовище розробки. Ось кілька прикладів популярних плагінів та їх внесок у розширення функціональності:

• *Code Glance* — додає міні-картку коду для зручної навігації файлів.

• *Rainbow Brackets* - розфарбовує дужки, роблячи код структурованішим.

• *Django Support* — забезпечує розширену підтримку для проектів Django.

• *GitToolBox* — покращує інтеграцію із системою контролю версій Git.

• *Markdown Support* — надає підсвічування синтаксису та зручний перегляд Markdown-файлів.

• *Database Tools* — забезпечує роботу з базами даних безпосередньо із середовища розробки.

• *PlantUML Integration* — дозволяє малювати діаграми UML у PyCharm.

Короткий довідник по PyCharm

- **Завантаження та встановлення** PyCharm
  - Завантаж з офіційного сайту: jetbrains.com/pycharm
  - Обери **Community Edition** (безкоштовна).
- **Створення першого проєкту**
  - Відкрий PyCharm → *New Project*
  - Вибери папку, де зберігатимеш файли.
  - Встанови **Virtual Environment (venv)** (PyCharm запропонує сам). Це дає незалежні бібліотеки для кожного проєкту.
- **Створення першого файлу**
  - Правий клік по проєкту → *New* → *Python File*.
  - Напиши:
  - print("Hello, PyCharm!")
  - Запусти через зелений трикутник (праворуч угорі).
- **Основні інструменти**

Автодоповнення

- Просто почни писати, і PyCharm підкаже функції.
- Ctrl+Space → список можливих варіантів.

Форматування коду

- Ctrl+Alt+L → автоматично вирівнює відступи.
- Попереджає про помилки (червоне підкреслення).

Налагодження (Debug)

- Постав *breakpoint* (клік зліва від рядка коду).
- Запусти *Debug* замість *Run*.
- Можна крок за кроком дивитись, як змінюються змінні.

Робота з бібліотеками. Інсталяція пакетів

- File → Settings → Python Interpreter.
- Додаєш бібліотеки (наприклад, numpy, pandas).
- Або прямо в терміналі:
- pip install numpy

Корисні поради

Швидкі клавіші

- Shift двічі → пошук у всьому проекті.
- Ctrl+Shift+F10 → запустити файл.
- Alt+Enter → швидке виправлення помилок.
- Ctrl+Q → пояснення функції/методу.

БІБЛІОГРАФІЯ

1. https://mek.oszk.hu/08400/08435/08435.pdf, Gérard Swinner,Tanuljunk meg programozni Python nyelven.
2. https://szit.hu/doku.php?id=oktatas:programozas:elemi_adatszerkezetek
3. https://oszkdk.oszk.hu/storage/00/00/59/28/dd/1/Adonyi_Adatstrukturak.pdf
4. http://www.informatom.hu/sze/01/LGB_SZ001/Algoritmusok_%C3%A9s_adatstrukt%C3%BAr%C3%A1k.pdf
5. https://web.uni-miskolc.hu/~matip/algoritmusok/
6. https://aries.ektf.hu/~hz/pdf-tamop/pdf-xx/algo-adatmodell.pdf
7. https://www.mechmat.univ.kiev.ua/wp-content/uploads/2021/09/pidruchnyk-alhorytmy-i-struktury-danykh.pdf

Виробничо-практичне видання

Алгоритми та структури даних/ Algoritmusok és adatszerkezetek
Методичні вказівки до самостійної роботи з дісципліни
«Алгоритми та структури даних»
2025 р.

Затверджено до використання у навчальному процесі
на засіданні кафедри математики та інформатики ЗУІ ім. Ф.Ракоці ІІ
(протокол № 10 від «23» червня 2025 року)

Розглянуто та рекомендовано Навчально-методичною радою
Закарпатського угорського інституту імені Ференца Ракоці ІІ
(протокол № 7 від «26» серпня 2025 року)

Рекомендовано до видання в електронній формі (PDF)
рішенням Вченої ради Закарпатського угорського інституту імені Ференца Ракоці ІІ
(протокол № 8 від «28 » серпня 2025 року)

Підготовлено до видання в електронній формі (PDF) кафедрою математики та інформатики спільно з Видавничим відділом Закарпатського угорського інституту імені Ференца Ракоці ІІ (39 с., українською мовою)

Розробник методичних вказівок:

Йожеф ГОЛОВАЧ – професор кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, доктор технічних наук

Рецензенти:

Федір ГЕЧЕ – професор УжНУ, доктор техніічих наук, професор

Олександр ТИЛИЩАК – професор кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, доктор фізико-математичних наук, доцент

Відповідальні за випуск:

Каталін КУЧІНКА – завідувач кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, доцент, кандидат фізико-математичних наук

Олександр ДОБОШ – начальник Видавничого відділу ЗУІ ім. Ф.Ракоці ІІ

За зміст методичних вказівок відповідальність несе розробник.

Видавництво: Закарпатський угорський інститут імені Ференца Ракоці ІІ (адреса: пл. Кошута 6, м. Берегове, 90202. Електронна пошта: foiskola@kmf.uz.ua) *Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру видавців, виготовлювачів і розповсюджувачів видавничої продукції Серія ДК ____ від _____ 2024 року*

Шрифт «Times New Roman». Розмір сторінок методичних вказівок: А4 (210х297мм).