Міністерство освіти і науки України
Міжнародний гуманітарний університет
Кафедра інформаційних технологій

М.А. Мірошник

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Опорний конспект лекцій
для здобувачів вищої освіти,
які навчаються за спеціальностями
галузі «Інформаційні технології»

Одеса 2025

Затверджено Вченою Радою Міжнародного гуманітарного університету.
Протокол № 5 від 20 січня 2025 р.

Мірошник М.А. Алгоритми та структури даних. Опорний конспект лекцій для здобувачів вищої освіти, які навчаються за спеціальностями галузі «Інформаційні технології» [Електронне видання]. Кафедра інформаційних технологій Міжнародного гуманітарного університету. Одеса, 2025. 33 с.

Підготовка бакалавра в галузі інформаційних технологій потребує глибокого розуміння принципів побудови алгоритмів і організації даних, що лежать в основі будь-якої програмної системи. Ефективність, масштабованість і надійність програмного забезпечення визначаються не лише якістю реалізації коду, а й правильністю вибору алгоритмічних підходів та структур даних. Саме ці компоненти забезпечують оптимальне використання обчислювальних ресурсів, коректність обробки великих обсягів інформації та передбачуваність поведінки систем у різних умовах експлуатації. Дисципліна «Алгоритми та структури даних» формує теоретичну та практичну основу для свідомого проєктування програмних рішень, поєднуючи формальний аналіз складності з реалізацією ефективних методів опрацювання даних. Застосування формальних методів аналізу складності забезпечує обґрунтованість проєктних рішень в галузі інформаційних технологій.

ЗМІСТ

ТЕМА 1. ВСТУП ДО АЛГОРИТМІВ, СТРУКТУР ДАНИХ ТА АНАЛІЗУ СКЛАДНОСТІ

Лекція 1. Основи алгоритмізації та структур даних

Поняття алгоритму є фундаментальним для інформатики та програмної інженерії, оскільки саме алгоритми визначають спосіб розв'язування задач у формалізованому вигляді. Алгоритм є точно визначеною скінченною послідовністю дій, виконання яких приводить до отримання результату для певного класу задач. Важливою ознакою алгоритму є його формальність, тобто можливість однозначного трактування кожного кроку незалежно від виконавця. Алгоритм не допускає неоднозначностей і повинен бути описаний таким чином, щоб його виконання не вимагало інтуїтивного або творчого підходу.

Алгоритм розглядається як універсальна модель обчислення, яка може бути реалізована як програмно, так і апаратно. У контексті програмного забезпечення алгоритм є логічною основою програми, а мова програмування виступає лише засобом його реалізації. Це означає, що якість програмного забезпечення значною мірою визначається якістю використаних алгоритмів.

До основних властивостей алгоритму належать дискретність, визначеність, скінченність, результативність та масовість. Дискретність означає, що алгоритм складається з окремих кроків, які виконуються послідовно. Визначеність передбачає, що кожен крок алгоритму є однозначним і не допускає альтернативних трактувань. Скінченність означає, що алгоритм завершується за скінченну кількість кроків. Результативність гарантує отримання певного результату після завершення виконання алгоритму. Масовість полягає в тому, що алгоритм застосовується до цілого класу задач, а не до одного конкретного випадку.

У практиці програмування важливим є також поняття ефективності алгоритму. Ефективність визначається витратами ресурсів, зокрема часу виконання та обсягу пам'яті. Аналіз ефективності алгоритмів є окремою важливою задачею, яка дозволяє обирати оптимальні рішення при розробці програмного забезпечення.

Будь-який алгоритм оперує даними, які поділяються на вхідні та вихідні. Вхідні дані є початковою інформацією, що подається алгоритму для обробки. Вихідні дані є результатом виконання алгоритму. Важливою характеристикою алгоритму є коректність, яка означає, що для будь-яких допустимих вхідних даних алгоритм формує правильний результат. Вхідні дані можуть бути задані у різних формах: числовій, текстовій, структурованій. Вихідні дані також можуть мати різну структуру залежно від задачі.

У складних програмних системах вхідні та вихідні дані часто мають складну структуру, що потребує використання спеціальних підходів до їх організації. Саме це приводить до поняття абстрактного типу даних. Абстрактний тип даних є формалізованим описом структури даних разом із набором операцій, які можуть бути виконані над цими даними, без уточнення способу їх реалізації.

Абстрактний тип даних визначає, що можна робити з даними, але не визначає, як це реалізується. Наприклад, стек як абстрактний тип даних визначає операції додавання елемента, видалення елемента та перегляду верхнього елемента, але не визначає, чи буде він реалізований на основі масиву чи зв'язаного списку. Такий підхід дозволяє відокремити інтерфейс від реалізації, що є важливим принципом програмної інженерії.

Використання абстрактних типів даних сприяє модульності програмного забезпечення та спрощує процес його супроводження. Зміна внутрішньої реалізації не впливає на інші частини системи, якщо інтерфейс залишається незмінним. Це дозволяє оптимізувати окремі компоненти без ризику порушення цілісності системи.

Структури даних є конкретними реалізаціями способів організації та зберігання даних у пам’яті комп’ютера. Вони визначають, як дані розташовані, як до них здійснюється доступ і які операції можуть бути виконані ефективно. Класифікація структур даних може здійснюватися за різними ознаками.

За способом організації розрізняють лінійні та нелінійні структури даних. До лінійних структур належать масиви, списки, стеки та черги. Вони характеризуються послідовним розташуванням елементів. Нелінійні структури, такі як дерева та графи, дозволяють відображати складні зв’язки між елементами.

За способом зберігання розрізняють статичні та динамічні структури даних. Статичні структури мають фіксований розмір, визначений на етапі компіляції. Прикладом є масиви. Динамічні структури можуть змінювати свій розмір під час виконання програми. До них належать зв’язані списки, дерева та інші структури, що використовують динамічне виділення пам’яті.

Також структури даних класифікуються за способом доступу до елементів. Існують структури з прямим доступом, такі як масиви, де будь-який елемент може бути отриманий за індексом за сталий час. Існують структури з послідовним доступом, такі як зв’язані списки, де доступ до елемента вимагає проходження через попередні елементи.

Взаємозв’язок алгоритмів і структур даних є ключовим аспектом програмування. Алгоритми визначають, які операції необхідно виконати, а структури даних визначають, як ці операції будуть реалізовані. Вибір структури даних безпосередньо впливає на ефективність алгоритму. Наприклад, пошук елемента у масиві може бути реалізований за лінійний час, тоді як використання дерева пошуку дозволяє скоротити час до логарифмічного.

Існує принцип, відповідно до якого ефективність алгоритму часто визначається не лише його логікою, але й правильно обраною структурою даних. Це означає, що оптимізація програмного забезпечення часто полягає у зміні структури даних, а не самого алгоритму. У сучасній програмній інженерії цей підхід використовується для побудови масштабованих і продуктивних систем.

Алгоритми відіграють ключову роль у проєктуванні програмного забезпечення. На етапі аналізу задачі формується алгоритмічна модель розв’язку, яка визначає основні етапи обробки даних. На етапі проєктування обираються відповідні структури даних і визначаються способи їх взаємодії. На етапі реалізації алгоритм трансформується у програмний код.

Якість алгоритму впливає на всі характеристики програмного забезпечення, включаючи продуктивність, надійність і масштабованість. Неефективний алгоритм може призвести до перевантаження системи, збільшення часу виконання або надмірного використання пам’яті. Тому вибір алгоритмічних рішень є критично важливим на ранніх етапах розробки.

У сучасних програмних системах алгоритми використовуються не лише для обчислень, але й для організації взаємодії між компонентами, обробки подій, управління ресурсами та забезпечення безпеки. Наприклад, алгоритми маршрутизації використовуються

у мережевих системах, алгоритми шифрування – у системах захисту інформації, алгоритми планування – у багатозадачних операційних системах.

Особливе значення мають алгоритми у системах реального часу, де необхідно забезпечити виконання операцій у строго визначені часові межі. У таких системах навіть незначні відхилення у роботі алгоритму можуть призвести до критичних наслідків. Це підкреслює необхідність ретельного аналізу та тестування алгоритмів.

Таким чином, алгоритми та структури даних є взаємопов'язаними складовими процесу розробки програмного забезпечення. Алгоритми визначають логіку розв'язування задач, а структури даних забезпечують ефективне зберігання та обробку інформації. Їх правильне поєднання дозволяє створювати ефективні, надійні та масштабовані програмні системи, що відповідають сучасним вимогам до якості програмного забезпечення.

Лекція 2. Аналіз алгоритмів та оцінка їх ефективності

Аналіз алгоритмів є ключовим етапом у розробці програмного забезпечення, оскільки дозволяє формально оцінити ефективність обчислювальних процедур ще до їх практичної реалізації або незалежно від конкретної апаратної платформи. У процесі аналізу визначається, як змінюються витрати ресурсів при зростанні розміру вхідних даних. Основними ресурсами, що розглядаються, є час виконання та обсяг використовуваної пам'яті.

Часова складність алгоритму характеризує кількість елементарних операцій, які виконуються алгоритмом залежно від розміру вхідних даних. Просторова складність визначає обсяг пам'яті, необхідний для виконання алгоритму. Обидві характеристики є взаємопов'язаними, оскільки оптимізація за одним параметром часто призводить до погіршення іншого.

Аналіз алгоритмів базується на понятті базової операції. Базовою операцією вважається така операція, яка виконується найчастіше або визначає основну обчислювальну складність алгоритму. До таких операцій належать арифметичні обчислення, операції порівняння, доступ до елементів масиву або присвоювання значень. При аналізі алгоритму визначається кількість виконань базової операції як функція від розміру вхідних даних n.

У найпростішому випадку кількість базових операцій можна визначити шляхом прямого підрахунку. Наприклад, для алгоритму обчислення суми елементів масиву довжини n базова операція додавання виконується n разів. Таким чином, часова складність такого алгоритму є лінійною.

У більш складних алгоритмах кількість базових операцій може залежати від структури вкладених циклів або умов виконання. Наприклад, якщо алгоритм містить два вкладені цикли, кожен з яких виконується n разів, то загальна кількість базових операцій становить n^2. У цьому випадку говорять про квадратичну складність.

Для формального опису залежності між розміром вхідних даних і кількістю операцій використовується асимптотичний аналіз. Його метою є визначення поведінки функції складності при великих значеннях n, коли вплив констант та менш значущих членів стає неістотним. Це дозволяє порівнювати алгоритми незалежно від конкретної реалізації та апаратного забезпечення.

Основними інструментами асимптотичного аналізу є нотації O, Ω та Θ. Нотація O (велике O) використовується для оцінки верхньої межі складності алгоритму. Вона показує,

що функція складності не перевищує певного значення при достатньо великих n. Наприклад, запис $O(n^2)$ означає, що час виконання алгоритму зростає не швидше, ніж квадратична функція.

Нотація Ω (омега) використовується для оцінки нижньої межі складності. Вона показує, що алгоритм виконує не менше певної кількості операцій. Наприклад, Ω(n) означає, що алгоритм у найкращому випадку виконує не менше ніж лінійну кількість операцій.

Нотація Θ (тета) задає точну асимптотичну оцінку, коли верхня і нижня межі збігаються. Це означає, що функція складності алгоритму зростає з точністю до констант так само, як задана функція. Наприклад, Θ(n log n) означає, що алгоритм має складність порядку n log n як у верхній, так і в нижній межах.

$$T(n) = O(f(n))$$

У практиці аналізу алгоритмів часто розглядають три випадки: найгірший, середній та найкращий. Найгірший випадок визначає максимальний час виконання алгоритму для всіх можливих вхідних даних заданого розміру. Саме ця оцінка є найбільш важливою при проєктуванні систем із гарантованими вимогами до продуктивності. Найкращий випадок характеризує мінімальний час виконання, а середній випадок визначає очікувану поведінку алгоритму при випадкових даних.

Порівняння алгоритмів за ефективністю здійснюється на основі їх асимптотичних оцінок. Наприклад, алгоритм із складністю O(n) є більш ефективним, ніж алгоритм із складністю $O(n^2)$, при достатньо великих значеннях n. Водночас для малих значень n менш ефективний алгоритм може працювати швидше через менші константи або простішу реалізацію.

Особливу увагу слід приділяти алгоритмам з логарифмічною та квазілінійною складністю. Алгоритми зі складністю O(log n), такі як бінарний пошук, демонструють дуже повільне зростання часу виконання навіть при значному збільшенні розміру даних. Алгоритми зі складністю O(n log n), наприклад ефективні алгоритми сортування, забезпечують компроміс між швидкодією та універсальністю.

Розглянемо приклади аналізу ітераційних алгоритмів. Ітераційні алгоритми характеризуються використанням циклів, що повторюють певну послідовність дій. Кількість ітерацій визначає загальну складність алгоритму.

Найпростішим прикладом є алгоритм, що містить один цикл:

```
for i від 1 до n
   виконати базову операцію
```

У цьому випадку кількість виконань базової операції дорівнює n, тому складність алгоритму становить Θ(n).

Розглянемо алгоритм із вкладеними циклами:

```
for i від 1 до n
   for j від 1 до n
      виконати базову операцію
```

Тут базова операція виконується $n \times n$ разів, тобто n^2 разів. Отже, складність алгоритму становить $\Theta(n^2)$.

Більш складним є випадок, коли межі вкладених циклів залежать одна від одної:

```
for i від 1 до n
   for j від 1 до i
      виконати базову операцію
```

У цьому випадку кількість виконань базової операції дорівнює сумі $1 + 2 + \ldots + n$, що становить $n(n+1)/2$. Асимптотично це відповідає $\Theta(n^2)$, оскільки домінуючим членом є $n^2/2$.

$$1 + 2 + \ldots + n = n(n+1)/2$$

Іншим прикладом є алгоритм із логарифмічною складністю:

```
i = 1
while i < n
   i = i * 2
   виконати базову операцію
```

У цьому випадку значення i подвоюється на кожній ітерації, тому кількість ітерацій дорівнює $\log_2 n$. Відповідно, складність алгоритму становить $\Theta(\log n)$.

Аналіз ітераційних алгоритмів часто зводиться до визначення кількості повторень циклів та оцінки вкладеності. При цьому важливо враховувати не лише кількість ітерацій, але й складність операцій, що виконуються всередині циклу.

У реальних програмних системах алгоритми можуть мати складніші структури, включаючи умовні оператори, рекурсію та взаємодію з різними структурами даних. Проте базові принципи аналізу залишаються незмінними: необхідно визначити ключові операції, оцінити їх кількість і застосувати асимптотичні нотації для узагальнення результату.

Роль аналізу алгоритмів у проєктуванні програмного забезпечення є визначальною. Він дозволяє приймати обґрунтовані рішення щодо вибору алгоритмів і структур даних, забезпечувати необхідний рівень продуктивності та масштабованості системи. У сучасних умовах, коли обсяги даних постійно зростають, ефективність алгоритмів стає критичним фактором успішності програмних продуктів.

Таким чином, аналіз алгоритмів є необхідним інструментом інженера програмного забезпечення. Він забезпечує формальну основу для оцінки ефективності рішень, дозволяє порівнювати альтернативні підходи та обирати оптимальні алгоритми для конкретних задач.

Лекція 3. Методи побудови алгоритмів. Рекурсія та ітерація

Методи побудови алгоритмів визначають підходи до формалізації процесу розв’язування задач і є основою проєктування ефективного програмного забезпечення. Вибір методу побудови алгоритму впливає на структуру програми, її зрозумілість, ефективність і можливість подальшого супроводження. Серед базових підходів особливе місце займають ітерація та рекурсія, а також більш загальні методи, такі як розділяй і володарюй.

Ітерація є одним із найпоширеніших способів організації алгоритмів. Вона передбачає багаторазове виконання певної послідовності дій за допомогою циклів. Ітераційні алгоритми характеризуються явним контролем кількості повторень через лічильники або умови завершення. Основною перевагою ітерації є її ефективність з точки зору використання пам'яті, оскільки вона не потребує додаткових витрат на збереження стану викликів, як це відбувається у випадку рекурсії.

Рекурсія є методом побудови алгоритмів, при якому функція викликає сама себе для розв'язання підзадачі меншого розміру. Рекурсивний підхід дозволяє природно описувати задачі, які мають самоподібну структуру. До таких задач належать обробка деревоподібних структур, пошук у графах, сортування та інші.

Кожен рекурсивний алгоритм складається з двох обов'язкових компонентів: базового випадку та рекурсивного випадку. Базовий випадок визначає умову завершення рекурсії і повертає результат без подальших викликів. Рекурсивний випадок описує, як задача розбивається на підзадачі, і включає виклик функції для цих підзадач.

Розглянемо приклад обчислення факторіала числа n. Базовий випадок полягає у тому, що факторіал нуля дорівнює одиниці. Рекурсивний випадок визначається співвідношенням факторіал n дорівнює n, помноженому на факторіал n−1. Такий підхід дозволяє звести задачу до простішого випадку, поки не буде досягнуто базового значення.

$$n! = n * (n-1)!$$

Рекурсія реалізується за допомогою механізму стеку викликів. Кожен виклик функції додається до стеку, де зберігаються значення параметрів, локальні змінні та адреса повернення. При досягненні базового випадку стек починає розгортатися, і результати обчислень повертаються у зворотному порядку.

Стек викликів є критично важливим елементом виконання рекурсивних алгоритмів. Його використання пов'язане з додатковими витратами пам'яті, оскільки кожен виклик функції створює новий запис у стеку. У випадку глибокої рекурсії це може призвести до переповнення стеку, що є типовою помилкою при неправильному проєктуванні алгоритмів.

Порівняння рекурсії та ітерації є важливим аспектом аналізу алгоритмів. Ітераційні алгоритми зазвичай є більш ефективними з точки зору використання пам'яті, оскільки вони не створюють додаткових записів у стеку. Водночас рекурсивні алгоритми часто є більш зрозумілими та компактними, особливо для задач із природною рекурсивною структурою.

У багатьох випадках рекурсивний алгоритм може бути перетворений в ітераційний і навпаки. Це перетворення базується на явному використанні структури даних, яка імітує стек викликів. Такий підхід дозволяє поєднати переваги обох методів.

Метод розділяй і володарюй є одним із найважливіших підходів до побудови алгоритмів. Його суть полягає у розбитті задачі на кілька підзадач меншого розміру, незалежному розв'язанні цих підзадач і об'єднанні отриманих результатів. Цей метод широко використовується у сортуванні, пошуку та обробці даних.

Класичним прикладом застосування методу розділяй і володарюй є алгоритм сортування злиттям. Задача сортування масиву розбивається на дві підзадачі сортування підмасивів, після чого результати об'єднуються у відсортований масив. Такий підхід забезпечує високу ефективність і передбачувану складність.

Рекурсія є природним інструментом реалізації методу розділяй і володарюй. Кожен рівень рекурсії відповідає новому рівню розбиття задачі. Це дозволяє будувати алгоритми з чіткою ієрархічною структурою.

Аналіз складності рекурсивних алгоритмів має свої особливості. На відміну від ітераційних алгоритмів, де складність визначається кількістю виконань циклів, у рекурсивних алгоритмах необхідно враховувати кількість рекурсивних викликів і складність кожного з них.

Для опису складності рекурсивних алгоритмів використовуються рекурентні співвідношення. Рекурентне співвідношення визначає залежність часу виконання алгоритму від розміру вхідних даних через час виконання підзадач. Наприклад, для алгоритму розділяй і володарюй співвідношення може мати вигляд:

$$T(n) = 2Tt(n/2) + n$$

Це співвідношення означає, що задача розбивається на дві підзадачі розміру n/2, а також виконується додаткова робота лінійної складності для об’єднання результатів.

Розв’язання рекурентних співвідношень дозволяє визначити асимптотичну складність алгоритму. У наведеному прикладі складність становить $\Theta(n \log n)$, що є типовим для ефективних алгоритмів сортування.

Існують різні методи розв’язання рекурентних співвідношень, зокрема метод підстановки, метод ітерацій та майстер-теорема. Вибір методу залежить від структури рекурентного співвідношення.

Аналіз рекурсивних алгоритмів також включає оцінку глибини рекурсії. Глибина рекурсії визначає максимальну кількість одночасно активних викликів функції і безпосередньо впливає на використання пам’яті. Для алгоритмів типу розділяй і володарюй глибина рекурсії зазвичай є логарифмічною.

Важливим аспектом є також оптимізація рекурсії. У деяких випадках компілятори можуть виконувати оптимізацію хвостової рекурсії, коли рекурсивний виклик є останньою операцією функції. Це дозволяє зменшити використання стеку і наблизити рекурсивний алгоритм до ітераційного за ефективністю.

Рекурсивні алгоритми широко використовуються у задачах обробки структур даних, таких як дерева і графи. Наприклад, обхід дерева природно реалізується за допомогою рекурсії, оскільки кожен вузол обробляється аналогічно своїм нащадкам. Це дозволяє створювати компактні та зрозумілі реалізації складних алгоритмів.

У проєктуванні програмного забезпечення вибір між рекурсією та ітерацією визначається вимогами до ефективності, обмеженнями пам’яті та складністю задачі. Рекурсія є доцільною для задач із природною ієрархічною структурою, тоді як ітерація є більш придатною для задач із чітко визначеною кількістю повторень.

Таким чином, методи побудови алгоритмів, зокрема рекурсія та ітерація, є основними інструментами інженера програмного забезпечення. Їх правильне використання дозволяє створювати ефективні, масштабовані та зрозумілі алгоритмічні рішення. Метод розділяй і володарюй забезпечує систематичний підхід до розв’язування складних задач, а аналіз складності рекурсивних алгоритмів дозволяє оцінити їх ефективність і забезпечити відповідність вимогам сучасних програмних систем.

ТЕМА 2. ЛІНІЙНІ СТРУКТУРИ ДАНИХ

Лекція 4. Масив як структура даних. Організація та ефективність операцій

Масив є однією з базових структур даних, яка широко використовується у програмуванні завдяки своїй простоті, ефективності доступу до елементів і передбачуваності поведінки. Масив визначається як впорядкована сукупність елементів одного типу, що розміщені у пам’яті послідовно та адресуються за індексом. Така організація забезпечує ефективний прямий доступ до будь-якого елемента.

Основною характеристикою масиву є спосіб розміщення елементів у пам’яті. Елементи масиву зберігаються у суміжних комірках пам’яті, що дозволяє обчислювати адресу довільного елемента за формулою, яка враховує базову адресу масиву та розмір елемента. Завдяки цьому доступ до елементів масиву здійснюється за сталий час незалежно від його розміру.

$$a_i = base + i * size$$

Така властивість робить масиви особливо ефективними для задач, де потрібен частий доступ до елементів за індексом, наприклад, при реалізації таблиць, матриць або буферів даних.

Залежно від способу організації пам’яті розрізняють статичні та динамічні масиви. Статичні масиви мають фіксований розмір, який визначається на етапі компіляції або під час створення структури. Після виділення пам’яті розмір такого масиву не може бути змінений. Це забезпечує простоту реалізації та відсутність додаткових накладних витрат на керування пам’яттю.

Динамічні масиви, на відміну від статичних, дозволяють змінювати свій розмір під час виконання програми. Вони реалізуються з використанням механізмів динамічного виділення пам’яті. Типовим прикладом є динамічні контейнери, які автоматично збільшують свій розмір при додаванні нових елементів. При цьому зазвичай використовується стратегія резервування додаткової пам’яті із запасом, що дозволяє зменшити кількість операцій перевиділення.

Організація динамічного масиву передбачає наявність двох параметрів: поточного розміру та ємності. Поточний розмір визначає кількість фактично збережених елементів, а ємність – обсяг виділеної пам’яті. Коли розмір досягає ємності, відбувається перевиділення пам’яті з більшим обсягом та копіювання елементів у нову область.

Операція доступу до елементів масиву є однією з найефективніших. Завдяки прямій адресації вона виконується за сталий час, що позначається як $\Theta(1)$. Це означає, що незалежно від кількості елементів у масиві час доступу до будь-якого з них залишається незмінним.

Вставлення та видалення елементів у масиві мають інші характеристики. Якщо вставлення відбувається у кінець масиву, то для динамічного масиву ця операція може виконуватися за амортизований сталий час, за умови, що перевиділення пам’яті відбувається нечасто. У статичному масиві вставлення у кінець можливе лише за наявності вільного місця.

Вставлення елемента у довільну позицію масиву потребує зсуву всіх наступних елементів на одну позицію вправо. Це призводить до лінійної складності операції, оскільки

кількість переміщень пропорційна кількості елементів після позиції вставлення. Аналогічно, видалення елемента з довільної позиції потребує зсуву елементів уліво, що також має лінійну складність.

$$T(n) = n$$

Таким чином, масиви є ефективними для операцій доступу, але менш ефективними для операцій вставлення та видалення у середині структури.

Порівняння статичних і динамічних масивів дозволяє оцінити їх переваги та обмеження. Статичні масиви мають мінімальні накладні витрати на керування пам’яттю та забезпечують максимальну ефективність доступу. Водночас їх основним недоліком є неможливість зміни розміру, що обмежує їх застосування у задачах із невизначеним обсягом даних.

Динамічні масиви забезпечують гнучкість у роботі з даними, дозволяючи змінювати розмір структури під час виконання програми. Однак це досягається за рахунок додаткових витрат пам’яті та часу на перевиділення та копіювання елементів. При цьому амортизований аналіз показує, що середній час додавання елемента у кінець залишається сталим, що робить такі структури практичними для більшості застосувань.

З точки зору використання пам’яті масиви мають високу ефективність, оскільки не потребують додаткових службових структур для зберігання зв’язків між елементами. Усі елементи розташовані компактно, що сприяє ефективному використанню кеш-пам’яті процесора. Це є важливою перевагою масивів порівняно з динамічними структурами, такими як зв’язані списки.

Проте динамічні масиви можуть мати певний надлишок пам’яті через резервування додаткового простору. Цей надлишок є компромісом між ефективністю операцій вставлення та використанням пам’яті. У практичних реалізаціях зазвичай використовується стратегія подвоєння ємності, що дозволяє забезпечити амортизовану ефективність.

Важливим аспектом використання масивів є їх роль у реалізації інших структур даних. Наприклад, масиви використовуються для побудови стеків, черг, хеш-таблиць та інших структур. У таких випадках властивості масиву визначають ефективність відповідних операцій.

У контексті проєктування програмного забезпечення вибір між масивами та іншими структурами даних визначається характером задачі. Якщо основною операцією є доступ до елементів за індексом, масив є оптимальним вибором. Якщо ж необхідні часті вставлення та видалення елементів у середині структури, доцільніше використовувати альтернативні структури.

Масиви також відіграють важливу роль у чисельних обчисленнях, обробці сигналів, комп’ютерній графіці та інших галузях, де необхідна робота з великими обсягами однорідних даних. Їх ефективність і простота роблять їх незамінним інструментом у багатьох прикладних задачах.

Таким чином, масив як структура даних є базовим елементом алгоритмічних рішень. Його властивості визначають ефективність доступу до даних, а також впливають на складність операцій модифікації. Розуміння особливостей статичних і динамічних масивів дозволяє обґрунтовано обирати структури даних у процесі розробки програмного забезпечення та забезпечувати оптимальне використання ресурсів системи.

Лекція 5. Зв’язні списки як динамічні структури даних

Зв’язні списки є базовою динамічною структурою даних, яка використовується для організації послідовностей елементів із можливістю ефективної модифікації структури під час виконання програми. На відміну від масивів, де елементи розташовані у суміжних ділянках пам’яті, зв’язні списки складаються з окремих вузлів, які можуть бути розміщені у довільних областях пам’яті та зв’язані між собою за допомогою посилань.

Кожен вузол зв’язного списку містить дві складові: інформаційне поле, у якому зберігається значення елемента, та службове поле, що містить адресу наступного вузла. Така організація дозволяє будувати структури змінного розміру без необхідності перевиділення пам’яті для всієї структури, як це відбувається у випадку масивів.

Однозв’язний список є найпростішим варіантом зв’язної структури. У ньому кожен вузол містить лише одне посилання – на наступний елемент списку. Перший вузол списку називається головою, а останній вузол містить посилання, що вказує на відсутність наступного елемента. Доступ до елементів у такому списку здійснюється послідовно, починаючи з голови.

Організація пам’яті однозв’язного списку базується на динамічному виділенні пам’яті для кожного вузла. Це означає, що вузли можуть бути розташовані у різних ділянках пам’яті, що знижує ефективність використання кешу процесора порівняно з масивами. Водночас така організація забезпечує високу гнучкість у зміні структури списку.

Двозв’язний список розширює можливості однозв’язного списку шляхом додавання другого посилання у кожному вузлі – на попередній елемент. Це дозволяє здійснювати двонапрямлений обхід списку, що є важливим для багатьох алгоритмів. Наявність посилання на попередній вузол спрощує операції видалення та вставлення, оскільки не потрібно додатково шукати попередній елемент.

Організація пам’яті двозв’язного списку передбачає більші витрати пам’яті порівняно з однозв’язним, оскільки кожен вузол містить два посилання замість одного. Проте це забезпечує більшу гнучкість у роботі зі структурою.

Основними операціями над зв’язними списками є вставлення, видалення та пошук елементів. Операція вставлення у список може бути реалізована ефективно, якщо відома позиція вставлення. Наприклад, вставлення нового вузла після заданого елемента виконується шляхом зміни посилань, що потребує сталого часу.

$$T(n) = 1$$

Це означає, що вставлення у зв’язному списку не залежить від його розміру, на відміну від масиву, де необхідно виконувати зсув елементів.

Видалення елемента зі списку також може бути виконане за сталий час, якщо є доступ до відповідного вузла або до попереднього вузла. У двозв’язному списку видалення спрощується завдяки наявності посилання на попередній елемент, що дозволяє виконати операцію без додаткового пошуку.

Операція пошуку у зв’язному списку виконується шляхом послідовного перегляду елементів, починаючи з голови. У найгіршому випадку необхідно переглянути всі елементи списку, що призводить до лінійної складності.

$$T(n) = n$$

Це є суттєвим недоліком зв’язних списків порівняно з масивами, де доступ до елемента за індексом виконується за сталий час.

Порівняння масивів і зв’язних списків за складністю операцій дозволяє визначити області їх ефективного застосування. Масиви забезпечують швидкий доступ до елементів за індексом, що робить їх оптимальними для задач, де домінує операція читання. Водночас вставлення та видалення елементів у середині масиву є дорогими операціями через необхідність зсуву елементів.

Зв’язні списки, навпаки, забезпечують ефективні операції вставлення та видалення, але мають обмеження щодо доступу до елементів. Відсутність прямого доступу означає, що будь-який доступ до довільного елемента потребує послідовного проходження списку.

З точки зору використання пам’яті масиви є більш ефективними, оскільки не потребують додаткових полів для зберігання посилань. Зв’язні списки мають додаткові витрати пам’яті на зберігання посилань, що може бути суттєвим при великій кількості елементів.

Важливим аспектом є також локальність даних. Масиви забезпечують високу локальність, оскільки елементи розташовані у суміжних комірках пам’яті. Це дозволяє ефективно використовувати кеш-пам’ять процесора. У зв’язних списках елементи можуть бути розташовані довільно, що знижує ефективність кешування.

У практиці програмної інженерії вибір між масивами та зв’язними списками визначається характером задачі. Якщо необхідна висока швидкість доступу до елементів і структура даних є відносно статичною, доцільно використовувати масиви. Якщо ж структура даних часто змінюється, а операції вставлення та видалення є критичними, зв’язні списки є більш доцільним вибором.

Зв’язні списки також використовуються як основа для реалізації інших структур даних, таких як стеки, черги та деякі види графів. Їх гнучкість дозволяє будувати складні структури з мінімальними обмеженнями.

Особливим випадком є циклічні списки, де останній елемент містить посилання на перший. Така організація використовується у задачах, де необхідна циклічна обробка даних. Іншим варіантом є списки з фіктивним вузлом, які спрощують реалізацію алгоритмів за рахунок усунення граничних випадків.

Таким чином, зв’язні списки є важливою альтернативою масивам у задачах, де необхідна гнучка структура даних. Їх використання дозволяє ефективно виконувати операції модифікації, але вимагає врахування обмежень щодо доступу та використання пам’яті. Розуміння відмінностей між масивами та списками є необхідною умовою для обґрунтованого вибору структур даних у процесі розробки програмного забезпечення.

Лекція 6. Стек і черга як абстрактні типи даних

Стек і черга є базовими абстрактними типами даних, що широко використовуються у програмній інженерії для організації обробки інформації. Вони визначають не лише спосіб зберігання даних, але й порядок доступу до них, що безпосередньо впливає на логіку алгоритмів. Абстрактний тип даних у цьому контексті задає набір допустимих операцій та їх семантику без прив’язки до конкретної реалізації.

Стек є структурою даних, яка реалізує принцип LIFO – останній доданий елемент видаляється першим. Це означає, що доступ до елементів здійснюється лише з одного кінця

структури, який називається вершиною стеку. Основними операціями стеку є додавання елемента на вершину та видалення елемента з вершини. Додатково використовується операція перегляду верхнього елемента без його видалення.

Принцип LIFO визначає характер використання стеку в алгоритмах. Він природно відповідає задачам, де необхідно обробляти дані у зворотному порядку їх надходження. Стек широко застосовується для організації викликів функцій, обробки вкладених структур та реалізації алгоритмів обходу.

LIFO: Last In First Out

Черга є структурою даних, яка реалізує принцип FIFO – перший доданий елемент видаляється першим. У черзі елементи додаються з одного кінця (хвоста) і видаляються з іншого (голови). Основними операціями є додавання елемента у кінець та видалення елемента з початку.

Принцип FIFO відповідає природному порядку обслуговування у багатьох системах, наприклад, у задачах планування процесів, обробки запитів або моделювання потоків даних. Черги забезпечують справедливий порядок обробки без пріоритетів.

FIFO: First In First Out

Реалізація стеку і черги може здійснюватися на основі масивів або зв’язних списків. Вибір конкретної реалізації впливає на ефективність операцій та використання пам’яті.

Реалізація стеку на основі масиву передбачає використання індексу, який вказує на вершину стеку. Додавання елемента полягає у збільшенні індексу та записі значення у відповідну позицію. Видалення елемента здійснюється шляхом зменшення індексу. Така реалізація забезпечує сталу складність основних операцій.

Реалізація стеку на основі зв’язного списку передбачає використання голови списку як вершини стеку. Додавання нового елемента здійснюється шляхом створення нового вузла та встановлення його посилання на попередню вершину. Видалення виконується шляхом перенесення голови на наступний вузол. Така реалізація також забезпечує сталу складність операцій, але дозволяє уникнути обмежень на розмір структури.

Черга на основі масиву зазвичай реалізується як циклічний буфер. Це дозволяє ефективно використовувати пам’ять і уникнути зсуву елементів при видаленні. Для цього використовуються два індекси: один для голови, інший для хвоста. При досягненні кінця масиву індекси повертаються на початок, що забезпечує циклічність структури.

Черга на основі зв’язного списку реалізується шляхом зберігання вказівників на перший і останній елементи. Додавання нового елемента виконується у кінець списку, а видалення – з початку. Такий підхід забезпечує сталу складність операцій і не потребує перевиділення пам’яті.

Дек є узагальненням черги, яке дозволяє виконувати вставлення та видалення елементів з обох кінців структури. Це робить дек більш універсальним, оскільки він може використовуватися як стек або як черга залежно від способу застосування. Дек може бути реалізований як на основі масиву, так і на основі двозв’язного списку.

Аналіз складності операцій для стеку і черги показує, що основні операції додавання та видалення виконуються за сталий час незалежно від кількості елементів.

$$T(n) = 1$$

Це є ключовою перевагою цих структур, оскільки вони забезпечують передбачувану ефективність незалежно від розміру даних.

Застосування стеку у алгоритмах є надзвичайно широким. Одним із найважливіших прикладів є організація рекурсивних викликів. Кожен виклик функції зберігається у стеку, що дозволяє відновити контекст виконання після завершення підзадачі. Це робить стек невід'ємною частиною механізму виконання програм.

Стек також використовується для перевірки коректності вкладених конструкцій, наприклад, дужок у виразах. При обробці кожного символу відкривальні дужки додаються у стек, а закривальні – видаляють відповідні елементи. Якщо після завершення обробки стек порожній, структура є коректною.

Ще одним важливим застосуванням є обчислення виразів у постфіксній формі. У цьому випадку стек використовується для зберігання операндів, а операції виконуються у порядку їх появи.

Черги широко використовуються у задачах обробки потоків даних та планування. Наприклад, у операційних системах черги застосовуються для організації виконання процесів. У мережевих системах черги використовуються для буферизації пакетів даних.

В алгоритмах пошуку черги використовуються для реалізації обходу графів у ширину. Такий підхід дозволяє досліджувати структуру рівень за рівнем, що є важливим для знаходження найкоротших шляхів у невагових графах.

Дек знаходить застосування у задачах, де необхідно гнучко змінювати порядок обробки даних. Наприклад, він може використовуватися для реалізації алгоритмів ковзного вікна або для підтримки двосторонніх операцій у буферних структурах.

У проєктуванні програмного забезпечення стек і черга виступають як базові будівельні блоки для більш складних структур і алгоритмів. Вони забезпечують ефективну організацію обробки даних і дозволяють реалізувати широкий спектр задач із передбачуваною складністю.

Вибір між реалізацією на основі масиву або списку визначається вимогами до системи. Масиви забезпечують кращу локальність даних і ефективне використання кешу, але мають обмеження на розмір або потребують перевиділення пам'яті. Списки забезпечують гнучкість і відсутність обмежень на розмір, але мають більші накладні витрати пам'яті.

Таким чином, стек і черга як абстрактні типи даних є фундаментальними елементами алгоритмічного мислення. Вони визначають порядок обробки даних і забезпечують ефективну реалізацію широкого спектра алгоритмів. Їх правильне використання є важливою умовою створення продуктивного та надійного програмного забезпечення.

ТЕМА 3. АЛГОРИТМИ ПОШУКУ ТА СОРТУВАННЯ

Лекція 7. Алгоритми пошуку та їх ефективність

Пошук є однією з базових задач інформатики, яка полягає у знаходженні елемента з заданими характеристиками у множині даних. Формально задача пошуку визначається як процес встановлення відповідності між заданим ключем і елементами структури даних. Результатом пошуку може бути або позиція елемента, або повідомлення про його відсутність.

Постановка задачі пошуку включає визначення вхідних даних, умов пошуку та очікуваного результату. Вхідними даними є структура даних і ключ, який потрібно знайти. Вихідними даними є індекс або посилання на знайдений елемент. У разі відсутності елемента алгоритм повинен коректно повідомити про це. Важливою характеристикою задачі пошуку є структура даних, оскільки вона визначає можливі алгоритми та їх ефективність.

Найпростішим алгоритмом пошуку є лінійний пошук. Він передбачає послідовний перегляд усіх елементів структури даних до моменту знаходження потрібного елемента або завершення перегляду. Лінійний пошук не потребує спеціальних умов і може застосовуватися до будь-якої структури, що підтримує послідовний доступ.

Алгоритм лінійного пошуку реалізується шляхом ітерації по елементах масиву або списку та порівняння кожного елемента з шуканим ключем. Якщо знайдено відповідність, алгоритм завершується. Якщо досягнуто кінця структури, результатом є відсутність елемента.

Часова складність лінійного пошуку залежить від положення елемента. У найкращому випадку елемент знаходиться на першій позиції, і складність становить сталу величину. У найгіршому випадку необхідно перевірити всі елементи, що дає лінійну складність.

$$T(n) = n$$

Середня складність також є лінійною, оскільки в середньому перевіряється половина елементів. Лінійний пошук є простим у реалізації, але неефективним для великих обсягів даних.

Бінарний пошук є більш ефективним алгоритмом, який застосовується до відсортованих структур даних. Його суть полягає у послідовному звуженні області пошуку шляхом порівняння шуканого елемента з центральним елементом поточного діапазону. Якщо ключ менший за центральний елемент, пошук продовжується у лівій частині, інакше – у правій.

На кожному кроці бінарного пошуку розмір області пошуку зменшується вдвічі. Це забезпечує логарифмічну складність алгоритму, що є значно ефективнішим порівняно з лінійним пошуком.

$$T(n) = \log_2 n$$

Умовою застосування бінарного пошуку є наявність впорядкованої структури даних. Якщо дані не відсортовані, необхідно попередньо виконати сортування, що може суттєво вплинути на загальну ефективність. Крім того, бінарний пошук вимагає можливості

швидкого доступу до середнього елемента, що робить його придатним для масивів, але менш ефективним для зв’язних списків.

Порівняння лінійного та бінарного пошуку показує, що вибір алгоритму залежить від умов задачі. Лінійний пошук є універсальним і не потребує попередньої обробки даних, але має низьку ефективність для великих наборів. Бінарний пошук забезпечує високу швидкодію, але вимагає впорядкованості даних і структури з прямим доступом.

Важливим аспектом є також вартість підготовки даних. Якщо пошук виконується багаторазово, доцільно витратити ресурси на сортування і використовувати бінарний пошук. Якщо ж пошук виконується одноразово або структура даних часто змінюється, лінійний пошук може бути більш доцільним.

Структура даних відіграє ключову роль у виборі алгоритму пошуку. У масивах доступ до елементів здійснюється за сталий час, що дозволяє ефективно реалізувати бінарний пошук. У зв’язних списках доступ є послідовним, тому навіть для відсортованих списків бінарний пошук втрачає свою ефективність, оскільки необхідно проходити елементи для досягнення середини.

У більш складних структурах даних, таких як дерева або хеш-таблиці, пошук може виконуватися ще ефективніше. Наприклад, у збалансованих деревах пошук виконується за логарифмічний час, а у хеш-таблицях – за сталий час у середньому випадку. Це демонструє важливість правильного вибору структури даних для задач пошуку.

Аналіз алгоритмів пошуку також враховує характер розподілу даних. У випадку нерівномірного розподілу або наявності частих повторень можуть застосовуватися спеціалізовані алгоритми, що враховують ці особливості.

У проєктуванні програмного забезпечення алгоритми пошуку використовуються у різних компонентах системи, зокрема у базах даних, файлових системах, інформаційно-пошукових системах та мережевих сервісах. Ефективність пошуку безпосередньо впливає на продуктивність системи та швидкість обробки запитів.

Таким чином, задача пошуку є фундаментальною для інформатики, а вибір алгоритму визначається умовами задачі, структурою даних та вимогами до ефективності. Лінійний пошук забезпечує універсальність і простоту, тоді як бінарний пошук забезпечує високу швидкодію за умови впорядкованості даних. Розуміння цих алгоритмів і їх властивостей є необхідною складовою підготовки інженера програмного забезпечення.

Лекція 8. Прості алгоритми сортування та їх аналіз

Сортування є однією з базових задач обробки даних, яка полягає у впорядкуванні елементів за певним критерієм, найчастіше за зростанням або спаданням значення ключа. У програмній інженерії сортування використовується як підготовчий етап для подальших операцій, зокрема пошуку, агрегування та аналізу даних. Простота реалізації та наочність роблять прості алгоритми сортування важливими як для навчання, так і для застосування у задачах із невеликими обсягами даних.

До простих алгоритмів сортування належать сортування обміном, сортування вибором та сортування вставками. Вони мають схожу асимптотичну складність, але відрізняються за кількістю порівнянь, перестановок і стабільністю.

Сортування обміном, або бульбашкове сортування, базується на багаторазовому проходженні масиву з порівнянням сусідніх елементів і їх перестановкою у разі неправильного порядку. На кожній ітерації найбільший елемент "переміщується" у кінець масиву. Процес повторюється, поки масив не стане впорядкованим.

Кількість порівнянь у бульбашковому сортуванні визначається як сума порівнянь на кожному проході. У найгіршому випадку виконується приблизно $n(n-1)/2$ порівнянь. Кількість перестановок також може досягати цієї величини у випадку повністю невпорядкованого масиву.

$$n(n-1)/2$$

Часова складність бульбашкового сортування у найгіршому та середньому випадках становить $\Theta(n^2)$. У найкращому випадку, якщо використовується оптимізація з перевіркою на відсутність обмінів, складність може бути знижена до $\Theta(n)$.

Сортування вибором базується на пошуку мінімального елемента у невпорядкованій частині масиву та його переміщенні на відповідну позицію. На кожному кроці алгоритм знаходить найменший елемент і виконує одну перестановку.

Кількість порівнянь у сортуванні вибором є фіксованою і не залежить від початкового порядку елементів. Вона також становить приблизно $n(n-1)/2$. Однак кількість перестановок значно менша – не більше ніж $n-1$, оскільки на кожному кроці виконується лише одна перестановка.

Часова складність сортування вибором у всіх випадках є однаковою і становить $\Theta(n^2)$, оскільки алгоритм завжди виконує повний набір порівнянь незалежно від впорядкованості даних.

Сортування вставками базується на поступовому формуванні впорядкованої частини масиву. На кожному кроці новий елемент вставляється у правильну позицію серед уже впорядкованих елементів. Це досягається шляхом зсуву елементів, що більші за поточний.

Кількість порівнянь і перестановок у сортуванні вставками залежить від початкового стану масиву. У найгіршому випадку, коли масив впорядкований у зворотному порядку, виконується приблизно $n(n-1)/2$ порівнянь і така ж кількість зсувів. У найкращому випадку, коли масив уже впорядкований, алгоритм виконує лише $n-1$ порівнянь.

$$T(n) = n^2$$

Часова складність сортування вставками у найгіршому та середньому випадках становить $\Theta(n^2)$, а у найкращому – $\Theta(n)$. Це робить його більш ефективним для майже впорядкованих даних.

Важливою характеристикою алгоритмів сортування є стабільність. Алгоритм вважається стабільним, якщо він зберігає відносний порядок елементів із однаковими ключами. Це є важливим у випадках, коли елементи мають додаткові атрибути.

Сортування обміном і сортування вставками є стабільними алгоритмами, оскільки вони не змінюють порядок рівних елементів. Сортування вибором у класичній реалізації не є стабільним, оскільки при перестановці може порушуватися порядок однакових елементів.

Порівняння алгоритмів сортування за кількістю операцій показує, що хоча всі три алгоритми мають квадратичну складність, вони відрізняються за практичною ефективністю.

Сортування вибором виконує мінімальну кількість перестановок, що може бути важливим у випадках, коли операції запису є дорогими. Сортування вставками є більш ефективним для частково впорядкованих даних. Сортування обміном є найпростішим у реалізації, але зазвичай поступається іншим алгоритмам за ефективністю.

Аналіз часової складності дозволяє зробити висновок, що прості алгоритми сортування не є ефективними для великих обсягів даних. Їх використання обмежується невеликими масивами або випадками, коли простота реалізації має більший пріоритет, ніж швидкодія.

У практиці програмної інженерії прості алгоритми сортування часто використовуються як допоміжні компоненти більш складних алгоритмів. Наприклад, вони можуть застосовуватися для сортування малих підмасивів у гібридних алгоритмах або для оптимізації окремих етапів обробки даних.

Таким чином, сортування обміном, вибором і вставками є базовими алгоритмами, що дозволяють зрозуміти основні принципи впорядкування даних. Їх аналіз з точки зору кількості порівнянь, перестановок, стабільності та часової складності формує основу для вивчення більш ефективних алгоритмів сортування та обґрунтованого вибору методів обробки даних у програмному забезпеченні.

Лекція 9. Ефективні алгоритми сортування та принцип розділяй і володарюй

Ефективні алгоритми сортування відіграють ключову роль у сучасних програмних системах, оскільки дозволяють обробляти великі обсяги даних із прийнятними витратами часу. На відміну від простих алгоритмів із квадратичною складністю, ефективні алгоритми забезпечують квазілінійну складність і базуються на більш складних принципах організації обчислень. До таких алгоритмів належать сортування злиттям, швидке сортування та пірамідальне сортування.

Одним із базових підходів до побудови ефективних алгоритмів є принцип розділяй і володарюй. Його суть полягає у розбитті задачі на підзадачі меншого розміру, незалежному їх розв’язанні та подальшому об’єднанні результатів. У задачах сортування цей принцип дозволяє значно зменшити кількість операцій порівняння та перестановки елементів.

Сортування злиттям є класичним прикладом реалізації методу розділяй і володарюй. Алгоритм рекурсивно розбиває масив на дві частини, поки не буде досягнуто підмасивів розміру один, які вже є впорядкованими. Після цього відбувається етап злиття, під час якого два впорядковані підмасиви об’єднуються в один впорядкований масив.

Часова складність сортування злиттям визначається рекурентним співвідношенням:

$$T(n) = 2T(n/2) + n$$

Розв’язання цього співвідношення дає оцінку $\Theta(n \log n)$, що робить алгоритм ефективним для великих обсягів даних. Важливою перевагою сортування злиттям є його стабільність, що означає збереження відносного порядку рівних елементів.

Основним недоліком є додаткові витрати пам’яті, оскільки для злиття підмасивів необхідно використовувати допоміжну структуру. Це обмежує застосування алгоритму у системах із жорсткими обмеженнями на пам’ять.

Швидке сортування також базується на принципі розділяй і володарюй, але реалізує його іншим способом. Алгоритм обирає опорний елемент і розділяє масив на дві частини: елементи, менші за опорний, і елементи, більші за нього. Після цього підмасиви сортуються рекурсивно.

У середньому випадку швидке сортування має складність $\Theta(n \log n)$, що робить його одним із найшвидших алгоритмів на практиці. Однак у найгіршому випадку, коли вибір опорного елемента є невдалим, складність може зростати до $\Theta(n^2)$.

$$T(n) = n^2$$

Практична ефективність швидкого сортування пояснюється його високою локальністю даних та відсутністю необхідності у додатковій пам’яті. Це робить його одним із найпоширеніших алгоритмів сортування у стандартних бібліотеках програмування.

Пірамідальне сортування базується на використанні спеціальної структури даних – бінарної купи. Алгоритм складається з двох основних етапів: побудови купи та послідовного видалення максимального (або мінімального) елемента з відновленням властивостей купи.

Часова складність пірамідального сортування у всіх випадках становить $\Theta(n \log n)$, що робить його стабільним з точки зору продуктивності. На відміну від швидкого сортування, він не має випадків деградації до квадратичної складності.

Перевагою пірамідального сортування є відсутність необхідності у додатковій пам’яті, оскільки воно може бути реалізоване безпосередньо у масиві. Однак цей алгоритм не є стабільним і має меншу практичну ефективність через складніші операції доступу до елементів.

Порівняння ефективних алгоритмів сортування дозволяє виділити їх основні характеристики. Сортування злиттям забезпечує стабільність і передбачувану складність, але потребує додаткової пам’яті. Швидке сортування є найефективнішим у середньому випадку та широко використовується на практиці, але має ризик деградації. Пірамідальне сортування забезпечує гарантовану складність без додаткових витрат пам’яті, але поступається за швидкодією у практичних реалізаціях.

Вибір алгоритму сортування залежить від конкретних умов задачі. Якщо важливо забезпечити стабільність, доцільно використовувати сортування злиттям. Якщо пріоритетом є швидкодія і допустимий ризик найгіршого випадку, оптимальним вибором є швидке сортування. Якщо необхідно гарантувати стабільну складність без використання додаткової пам’яті, доцільно застосовувати пірамідальне сортування.

У сучасних програмних системах часто використовуються гібридні алгоритми, які поєднують переваги різних підходів. Наприклад, швидке сортування може доповнюватися іншими алгоритмами для обробки малих підмасивів або уникнення найгірших випадків.

Таким чином, ефективні алгоритми сортування є важливим інструментом оптимізації програмного забезпечення. Принцип розділяй і володарюй забезпечує систематичний підхід до розв’язування задач, а аналіз складності дозволяє обґрунтовано обирати алгоритми залежно від вимог до продуктивності, використання пам’яті та стабільності.

ТЕМА 4. НЕЛІНІЙНІ СТРУКТУРИ ДАНИХ

Лекція 10. Дерева як ієрархічні структури даних

Дерева є однією з фундаментальних нелінійних структур даних, що широко застосовуються у програмній інженерії для представлення ієрархічних залежностей між об'єктами. На відміну від лінійних структур, таких як масиви або списки, дерева дозволяють моделювати складні відношення підпорядкування, що є характерним для багатьох прикладних задач, зокрема організації файлових систем, синтаксичного аналізу програм, побудови індексів баз даних та представлення ієрархій.

Формально дерево визначається як зв'язна ациклічна структура, що складається з вузлів, між якими існують відношення батьківства. Один із вузлів є коренем дерева, і від нього починається вся структура. Кожен вузол, окрім кореня, має рівно одного батька та може мати нуль або більше нащадків.

Основними поняттями, що використовуються при описі дерев, є вузол, рівень, висота та піддерево. Вузол є елементом дерева, який містить дані та, залежно від реалізації, посилання на інші вузли. Рівень вузла визначається як кількість ребер на шляху від кореня до цього вузла. Корінь має рівень нуль або одиницю залежно від прийнятої конвенції. Висота дерева визначається як максимальна глибина вузлів, тобто довжина найдовшого шляху від кореня до листа.

Піддерево є частиною дерева, що складається з певного вузла та всіх його нащадків. Це поняття є ключовим при реалізації рекурсивних алгоритмів обробки дерев, оскільки будь-яке дерево можна розглядати як сукупність піддерев.

Бінарні дерева є окремим і дуже важливим випадком дерев, у яких кожен вузол має не більше двох нащадків. Ці нащадки зазвичай називаються лівим і правим піддеревами. Така обмеженість спрощує реалізацію алгоритмів і дозволяє ефективно використовувати структуру для пошуку, сортування та інших задач.

Бінарні дерева можуть мати різні властивості залежно від умов їх побудови. Наприклад, повне бінарне дерево є таким, у якому всі рівні, окрім, можливо, останнього, повністю заповнені. Ідеально збалансоване дерево має мінімальну можливу висоту, що забезпечує ефективний доступ до елементів.

Однією з ключових операцій над деревами є їх обхід, тобто систематичне відвідування всіх вузлів. Існує кілька основних способів обходу бінарного дерева, які визначають порядок обробки вузлів.

Прямий обхід передбачає спочатку обробку поточного вузла, потім лівого піддерева і, нарешті, правого піддерева. Такий підхід використовується для копіювання дерева або побудови його структурного представлення.

Симетричний обхід передбачає спочатку обробку лівого піддерева, потім поточного вузла і, нарешті, правого піддерева. Для бінарних дерев пошуку цей обхід дозволяє отримати елементи у відсортованому порядку, що є важливою властивістю.

Зворотний обхід передбачає спочатку обробку піддерев, а потім поточного вузла. Він використовується, наприклад, для звільнення пам'яті, зайнятої деревом, або для обчислення значень у вузлах, що залежать від нащадків.

Обхід дерева може бути реалізований як рекурсивно, так і ітераційно. Рекурсивна реалізація є більш природною, оскільки структура дерева безпосередньо відповідає

рекурсивному опису. Ітераційна реалізація потребує використання допоміжних структур, зокрема стеку або черги.

Реалізація дерев у пам’яті може здійснюватися різними способами. Найбільш поширеним є представлення за допомогою зв’язних структур, де кожен вузол містить дані та посилання на своїх нащадків. Для бінарного дерева це означає наявність двох посилань – на лівий і правий вузли.

Такий підхід забезпечує гнучкість і дозволяє створювати дерева довільної форми. Водночас він потребує додаткових витрат пам’яті на зберігання посилань і має меншу локальність даних порівняно з масивами.

Альтернативним способом є представлення дерева у вигляді масиву. Цей підхід ефективний для повних або майже повних бінарних дерев, де можна використовувати індексацію для визначення положення нащадків. Наприклад, для вузла з індексом i лівий нащадок має індекс 2i, а правий – 2i+1.

$$left = 2i, right = 2i + 1$$

Такий спосіб реалізації забезпечує високу ефективність доступу та хорошу локальність даних, але є неефективним для розріджених дерев, де значна частина масиву залишається невикористаною.

Вибір способу реалізації дерева залежить від характеру задачі. Якщо дерево має регулярну структуру і близьке до повного, доцільно використовувати масивне представлення. Якщо структура є динамічною та нерегулярною, більш придатною є реалізація на основі вузлів із посиланнями.

Дерева широко використовуються у різних областях програмування. У компіляторах вони застосовуються для побудови синтаксичних дерев, у базах даних – для організації індексів, у мережевих системах – для маршрутизації. Їх здатність ефективно представляти ієрархічні відношення робить їх незамінними у складних програмних системах.

Важливим аспектом є також балансування дерев, яке дозволяє підтримувати оптимальну висоту структури. Незбалансоване дерево може вироджуватися у лінійну структуру, що призводить до зниження ефективності операцій. Це підкреслює необхідність використання спеціалізованих алгоритмів для підтримки властивостей дерева.

Таким чином, дерева є потужним інструментом представлення та обробки даних. Розуміння їх структури, способів обходу та реалізації у пам’яті дозволяє ефективно застосовувати їх у різних задачах програмної інженерії та забезпечувати високу продуктивність програмних систем.

Лекція 11. Бінарні дерева пошуку та ефективність операцій

Бінарне дерево пошуку є однією з найважливіших структур даних, яка поєднує ієрархічну організацію з ефективними алгоритмами доступу до елементів. Його основною особливістю є впорядкованість вузлів, що забезпечується спеціальною властивістю: для кожного вузла всі елементи у лівому піддереві є меншими за значення вузла, а всі елементи у правому піддереві – більшими. Ця властивість дозволяє виконувати пошук, вставлення та видалення значно ефективніше, ніж у лінійних структурах даних.

Пошук у бінарному дереві пошуку виконується шляхом порівняння шуканого ключа зі значенням поточного вузла. Якщо ключ дорівнює значенню вузла, пошук завершується. Якщо ключ менший, пошук продовжується у лівому піддереві, якщо більший – у правому. Такий підхід дозволяє на кожному кроці відкидати половину елементів, що робить алгоритм аналогічним бінарному пошуку у масиві, але без необхідності попереднього зберігання даних у вигляді суцільної структури.

Часова складність пошуку залежить від висоти дерева. У випадку збалансованого дерева висота пропорційна логарифму від кількості елементів, що забезпечує ефективність операції.

$$T(n) = \log n$$

У найгіршому випадку, коли дерево вироджується у лінійну структуру, складність пошуку зростає до лінійної.

Операція вставлення у бінарне дерево пошуку виконується аналогічно пошуку. Спочатку знаходиться позиція, у якій має бути розміщений новий елемент, після чого створюється новий вузол і додається як лист. Вставлення не потребує перестановки інших елементів, що є суттєвою перевагою порівняно з масивами.

Складність вставлення також залежить від висоти дерева. У збалансованому випадку вона є логарифмічною, а у виродженому – лінійною.

Видалення вузла є більш складною операцією, оскільки необхідно зберегти властивість впорядкованості дерева. Існує три основні випадки видалення. Якщо вузол не має нащадків, він просто видаляється. Якщо вузол має одного нащадка, він замінюється цим нащадком. Якщо вузол має двох нащадків, необхідно знайти елемент, який замінить видалений вузол. Зазвичай використовується або найменший елемент у правому піддереві, або найбільший у лівому.

Операція видалення також має складність, що залежить від висоти дерева, і у збалансованому випадку становить логарифмічну величину.

Важливим фактором, що визначає ефективність усіх операцій у бінарному дереві пошуку, є його структура. Якщо елементи додаються у випадковому порядку, дерево має тенденцію бути відносно збалансованим. Якщо ж елементи додаються у впорядкованому вигляді, дерево може виродитися у лінійний список, що призводить до різкого погіршення ефективності.

$$T(n) = n$$

Це означає, що навіть ефективна структура даних може втратити свої переваги при неправильному використанні.

Поняття збалансованості дерева є ключовим для забезпечення стабільної ефективності. Збалансоване дерево характеризується тим, що висоти його піддерев не відрізняються більш ніж на певну величину. Це гарантує, що довжина шляху від кореня до будь-якого вузла залишається логарифмічною відносно кількості елементів.

Існують спеціальні типи бінарних дерев пошуку, які автоматично підтримують збалансованість, наприклад AVL-дерева або червоно-чорні дерева. У таких структурах після кожної операції вставлення або видалення виконуються додаткові перетворення, що забезпечують збереження властивостей дерева.

Балансування дерева дозволяє уникнути деградації продуктивності та забезпечує передбачувану складність операцій. Це є особливо важливим у системах, де обробляються великі обсяги даних або існують жорсткі вимоги до часу виконання.

У проєктуванні програмного забезпечення бінарні дерева пошуку використовуються для реалізації словників, множин, індексів та інших структур, де необхідно забезпечити швидкий доступ до елементів. Вони є компромісом між простотою реалізації та ефективністю, що робить їх універсальним інструментом.

Таким чином, бінарне дерево пошуку є потужною структурою даних, яка забезпечує ефективні операції пошуку, вставлення та видалення за умови підтримки збалансованості. Розуміння впливу структури дерева на складність операцій дозволяє правильно використовувати цю структуру та уникати ситуацій, що призводять до зниження продуктивності.

Лекція 12. Хеш-таблиці та ефективність доступу до даних

Хеш-таблиці є однією з найефективніших структур даних для організації швидкого доступу до інформації. Вони дозволяють виконувати операції пошуку, вставлення та видалення за сталий час у середньому випадку, що робить їх надзвичайно важливими у сучасних програмних системах. Основна ідея хеш-таблиці полягає у використанні функції відображення, яка перетворює ключ у індекс масиву.

Хеш-таблиця реалізується як масив фіксованого розміру, у якому зберігаються елементи. Для визначення позиції елемента використовується хеш-функція, яка обчислює індекс на основі значення ключа. Таким чином, замість послідовного пошуку по всій структурі, доступ до елемента здійснюється безпосередньо за обчисленим індексом.

Хеш-функція є критичним компонентом хеш-таблиці. Вона повинна забезпечувати рівномірний розподіл ключів по доступних позиціях масиву, щоб мінімізувати кількість конфліктів. Ідеальна хеш-функція повинна бути швидкою у обчисленні, детермінованою та давати різні значення для різних ключів.

$$h(k) = k \bmod m$$

Наведена формула є прикладом простої хеш-функції, де значення ключа береться за модулем розміру таблиці. Хоча така функція є простою, вона може призводити до нерівномірного розподілу, якщо ключі мають певні закономірності.

Однією з основних проблем при використанні хеш-таблиць є колізії. Колізія виникає у випадку, коли два різні ключі відображаються на одну й ту саму позицію у таблиці. Оскільки кількість можливих ключів зазвичай перевищує розмір таблиці, колізії є неминучими.

Існують різні методи розв’язання колізій. Одним із найпоширеніших є метод ланцюжків, при якому кожна позиція таблиці містить список елементів із однаковим хеш-значенням. У цьому випадку при виникненні колізії новий елемент додається до відповідного списку.

Іншим підходом є відкрита адресація, при якій усі елементи зберігаються безпосередньо у масиві. У разі колізії алгоритм шукає іншу вільну позицію за певним правилом. Існують різні стратегії відкритої адресації, зокрема лінійне пробування, квадратичне пробування та подвійне хешування.

Лінійне пробування передбачає послідовний перегляд наступних позицій масиву до знаходження вільного місця. Цей метод є простим, але може призводити до утворення кластерів, що знижує ефективність.

Квадратичне пробування використовує квадратичну залежність для визначення наступної позиції, що дозволяє зменшити скупчення елементів. Подвійне хешування застосовує другу хеш-функцію для визначення кроку зміщення, що забезпечує більш рівномірний розподіл.

Ефективність хеш-таблиці значною мірою визначається коефіцієнтом заповнення, який показує відношення кількості елементів до розміру таблиці. При збільшенні цього коефіцієнта зростає ймовірність колізій, що призводить до зниження ефективності.

$$a = n/m$$

де n – кількість елементів, m – розмір таблиці.

У середньому випадку операції пошуку та вставлення у хеш-таблиці виконуються за сталий час, що позначається як $\Theta(1)$. Це досягається за умови рівномірного розподілу ключів і низького коефіцієнта заповнення.

$$T(n) = 1$$

У найгіршому випадку, коли всі елементи потрапляють в одну позицію або утворюється довгий кластер, складність може зростати до лінійної.

$$T(n) = n$$

Аналіз ефективності хеш-таблиць показує, що їх продуктивність залежить від якості хеш-функції та обраного методу розв’язання колізій. Добре спроєктована хеш-таблиця забезпечує стабільну та передбачувану швидкодію навіть при значних обсягах даних.

У практичних реалізаціях часто використовується динамічне розширення таблиці, коли при досягненні певного коефіцієнта заповнення розмір масиву збільшується, а всі елементи перехешовуються. Це дозволяє підтримувати ефективність операцій на високому рівні.

Хеш-таблиці широко застосовуються у реалізації асоціативних масивів, кешів, баз даних, компіляторів та інших систем, де необхідний швидкий доступ до даних за ключем. Вони є основою багатьох сучасних алгоритмів і структур даних.

Таким чином, хеш-таблиці забезпечують один із найефективніших способів організації доступу до даних. Поняття хеш-функції, методи розв’язання колізій та аналіз ефективності дозволяють правильно проєктувати ці структури та використовувати їх у складних програмних системах з високими вимогами до продуктивності.

ТЕМА 5. ГРАФИ ТА АЛГОРИТМИ НА ГРАФАХ

Лекція 13. Графи як структура даних та способи їх подання

Графи є однією з найбільш універсальних структур даних, що використовуються для моделювання складних систем взаємозв'язків. Вони дозволяють формалізувати об'єкти та відношення між ними, що є характерним для широкого спектра задач: від комп'ютерних мереж і соціальних систем до транспортних моделей і алгоритмів маршрутизації. У програмній інженерії графи застосовуються для аналізу залежностей, оптимізації процесів і побудови складних алгоритмічних рішень.

Формально граф визначається як пара множин: множини вершин і множини ребер, що з'єднують ці вершини. Вершини представляють об'єкти, а ребра – зв'язки між ними. Така абстракція дозволяє описувати як прості, так і складні структури.

$$G = (V, E)$$

де V – множина вершин, E – множина ребер.

Залежно від властивостей зв'язків розрізняють орієнтовані та неорієнтовані графи. В орієнтованому графі кожне ребро має напрямок, тобто задає впорядковану пару вершин. Це означає, що зв'язок між вершинами є несиметричним. Такий тип графів використовується для моделювання процесів із визначеним напрямком, наприклад потоків даних або залежностей між завданнями.

У неорієнтованому графі ребра не мають напрямку, і зв'язок між вершинами є симетричним. Якщо існує ребро між двома вершинами, то перехід можливий у обох напрямках. Такі графи використовуються для моделювання взаємних зв'язків, наприклад у соціальних мережах або мережах доріг без одностороннього руху.

Важливою характеристикою графів є спосіб їх представлення у пам'яті. Найбільш поширеними є матриця суміжності та список суміжності.

Матриця суміжності є двовимірним масивом розміру n × n, де n – кількість вершин графа. Елемент матриці визначає наявність або відсутність ребра між відповідними вершинами. У випадку орієнтованого графа матриця є несиметричною, а у неорієнтованого – симетричною.

$$A[i][j] = 1 \text{ якщо існує ребро}$$

Перевагою матриці суміжності є простота реалізації та швидкий доступ до інформації про наявність ребра, який виконується за сталий час. Однак цей підхід має значні витрати пам'яті, особливо для розріджених графів, де кількість ребер значно менша за максимально можливу.

Список суміжності є більш ефективним способом представлення графів, особливо для розріджених структур. У цьому випадку для кожної вершини зберігається список суміжних вершин. Така структура дозволяє зберігати лише існуючі ребра, що значно зменшує витрати пам'яті.

Доступ до сусідів вершини у списку суміжності виконується ефективно, але перевірка наявності конкретного ребра може потребувати перебору елементів списку. Таким чином,

вибір способу представлення залежить від характеру задачі та співвідношення між кількістю вершин і ребер.

Основні характеристики графів визначають їх структуру та впливають на вибір алгоритмів обробки. Однією з ключових характеристик є ступінь вершини. У неорієнтованому графі ступінь визначається як кількість ребер, інцидентних вершині. В орієнтованому графі розрізняють вхідний і вихідний ступені.

Іншою важливою характеристикою є шлях у графі, тобто послідовність вершин, з'єднаних ребрами. Якщо існує шлях між двома вершинами, говорять, що вони є досяжними. Граф називається зв'язним, якщо між будь-якими двома вершинами існує шлях.

Цикл у графі є шляхом, який починається і закінчується в одній і тій самій вершині. Наявність циклів визначає багато властивостей графа та впливає на вибір алгоритмів обробки. Наприклад, у деяких задачах необхідно перевірити відсутність циклів для забезпечення коректності структури.

Важливим поняттям є також компоненти зв'язності – підграфи, у яких будь-які дві вершини зв'язані шляхом. У орієнтованих графах розрізняють сильну та слабку зв'язність залежно від напрямків ребер.

Ще однією характеристикою є щільність графа, яка визначається відношенням кількості ребер до максимально можливої кількості. Щільні графи мають велику кількість ребер, тоді як розріджені – малу. Це безпосередньо впливає на вибір способу представлення та алгоритмів обробки.

У практиці програмної інженерії графи використовуються для вирішення широкого спектра задач. Наприклад, у мережевих системах вони застосовуються для моделювання топології мережі, у системах керування – для аналізу залежностей, у компіляторах – для побудови графів викликів.

Розуміння структури графів і способів їх представлення є необхідною умовою для ефективного використання алгоритмів, таких як пошук у глибину, пошук у ширину, знаходження найкоротших шляхів та інші. Вибір правильної структури даних дозволяє суттєво вплинути на продуктивність системи.

Таким чином, граф як структура даних є потужним інструментом моделювання складних взаємозв'язків. Орієнтовані та неорієнтовані графи дозволяють описувати різні типи відношень, а матриця суміжності та список суміжності забезпечують різні способи реалізації у пам'яті. Розуміння основних характеристик графів є базою для подальшого вивчення алгоритмів обробки графових структур і їх застосування у програмному забезпеченні.

Лекція 14. Обхід графів: пошук у глибину та пошук у ширину

Обхід графів є базовою операцією, яка полягає у систематичному відвідуванні вершин з метою аналізу структури, пошуку шляхів або визначення властивостей графа. Алгоритми обходу лежать в основі багатьох задач, зокрема визначення компонент зв'язності, побудови маршрутів, перевірки досяжності та аналізу залежностей. Найбільш поширеними алгоритмами обходу є пошук у глибину та пошук у ширину.

Пошук у глибину базується на ідеї максимально можливого заглиблення у граф перед переходом до інших гілок. Алгоритм починається з вибраної вершини і рекурсивно

переходить до суміжних вершин, поки не досягне вершини, з якої немає невідвіданих сусідів. Після цього відбувається повернення до попередньої вершини і продовження обходу.

Реалізація пошуку у глибину може здійснюватися як рекурсивно, так і з використанням стеку. Рекурсивна реалізація є більш природною, оскільки структура викликів відповідає ієрархії обходу. Використання стеку дозволяє явно контролювати порядок відвідування вершин.

Пошук у ширину, на відміну від пошуку у глибину, виконує обхід графа рівень за рівнем. Алгоритм починається з початкової вершини, після чого відвідує всі її сусіди, потім сусідів цих сусідів і так далі. Такий підхід забезпечує знаходження найкоротших шляхів у невагових графах.

Реалізація пошуку у ширину базується на використанні черги. Вершини додаються у чергу у порядку їх виявлення, а обробка відбувається у порядку надходження. Це забезпечує правильну послідовність обходу.

Під час виконання алгоритмів обходу формується дерево обходу, яке відображає структуру переходів між вершинами. Кожне ребро цього дерева відповідає переходу від однієї вершини до іншої під час обходу. Дерево обходу є підграфом початкового графа і дозволяє аналізувати його структуру.

У пошуку у глибину дерево обходу відображає глибину вкладеності переходів, тоді як у пошуку у ширину воно має рівневу структуру. Це визначає різні властивості цих алгоритмів і їх застосування.

Компоненти зв’язності є важливою характеристикою графа, яка визначає його структуру. Компонента зв’язності – це максимальна підмножина вершин, у якій будь-які дві вершини з’єднані шляхом. Для знаходження компонент зв’язності використовується багаторазове застосування алгоритмів обходу.

Алгоритм визначення компонент зв’язності полягає у наступному. Обирається невідвідана вершина, після чого виконується обхід графа, який відвідує всі вершини, досяжні з цієї вершини. Усі ці вершини утворюють одну компоненту зв’язності. Процес повторюється для інших невідвіданих вершин.

У орієнтованих графах поняття зв’язності ускладнюється, і розрізняють сильну та слабку зв’язність. Сильна зв’язність означає, що для будь-якої пари вершин існують шляхи в обох напрямках. Алгоритми обходу також використовуються для визначення таких структур.

Аналіз складності алгоритмів обходу базується на кількості відвіданих вершин і ребер. У загальному випадку кожна вершина відвідується один раз, а кожне ребро розглядається не більше двох разів.

$$T(V, E) = V + E$$

де V – кількість вершин, E – кількість ребер.

Це означає, що алгоритми пошуку у глибину та пошуку у ширину мають лінійну складність відносно розміру графа. Така ефективність робить їх придатними для роботи з великими графами.

Вибір способу представлення графа впливає на практичну ефективність алгоритмів обходу. При використанні списку суміжності алгоритми працюють оптимально, оскільки обробляються лише існуючі ребра. У випадку матриці суміжності необхідно перевіряти всі можливі ребра, що збільшує витрати часу.

Пошук у глибину широко використовується для вирішення задач, пов’язаних із аналізом структури графа, наприклад, виявлення циклів, топологічного сортування або побудови компонент зв’язності. Пошук у ширину застосовується для знаходження найкоротших шляхів у невагових графах і для аналізу рівневої структури.

Важливою особливістю алгоритмів обходу є необхідність відмічати відвідані вершини, щоб уникнути зациклення. Це досягається за допомогою допоміжних структур даних, наприклад масиву або множини відвіданих вершин.

У практиці програмної інженерії алгоритми обходу графів використовуються у широкому спектрі задач, включаючи маршрутизацію, аналіз соціальних мереж, оптимізацію процесів і побудову залежностей між компонентами системи.

Таким чином, обхід графів є фундаментальною операцією, що дозволяє ефективно досліджувати структуру графа. Пошук у глибину та пошук у ширину забезпечують різні підходи до обходу, кожен із яких має свої переваги та області застосування. Аналіз їх складності показує, що ці алгоритми є ефективними і придатними для використання у складних програмних системах.

Лекція 15. Алгоритми на графах та їх застосування у програмній інженерії

Графові алгоритми є важливим класом алгоритмів, що застосовуються для розв’язування задач, пов’язаних із аналізом складних взаємозв’язків між об’єктами. Вони широко використовуються у мережевих системах, логістиці, аналізі даних, компіляторах, інформаційних системах та багатьох інших галузях. Основними задачами, що розв’язуються на графах, є пошук найкоротших шляхів, побудова мінімальних остовних дерев та оптимізація процесів за допомогою жадібних алгоритмів.

Пошук найкоротших шляхів є однією з ключових задач у теорії графів. Вона полягає у знаходженні шляху між двома вершинами з мінімальною сумарною вагою ребер. У невагових графах ця задача розв’язується за допомогою пошуку у ширину, який забезпечує знаходження найкоротшого шляху за кількістю ребер.

У зважених графах застосовуються більш складні алгоритми. Одним із найвідоміших є алгоритм Дейкстри, який дозволяє знаходити найкоротші шляхи від однієї вершини до всіх інших у графі з невід’ємними вагами ребер. Алгоритм базується на поступовому розширенні множини вершин із відомими найкоротшими відстанями.

$$d(v) = \min(d(u) + w(u, v))$$

де $d(v)$ – найкоротша відстань до вершини v, $w(u, v)$ – вага ребра між вершинами.

Складність алгоритму Дейкстри залежить від реалізації та використаних структур даних. У випадку використання пріоритетної черги вона становить $\Theta((V + E) \log V)$, що забезпечує ефективність навіть для великих графів.

Іншою важливою задачею є побудова мінімального остовного дерева. Остовне дерево – це підграф, який містить усі вершини графа та мінімальну кількість ребер, необхідних для їх з’єднання. Мінімальне остовне дерево має мінімальну сумарну вагу серед усіх можливих остовних дерев.

Для побудови мінімального остовного дерева використовуються алгоритми Прима та Крускала. Алгоритм Прима починається з довільної вершини і поступово додає ребра з мінімальною вагою, що з'єднують вже побудовану частину дерева з іншими вершинами. Алгоритм Крускала, навпаки, розглядає ребра у порядку зростання їх ваги і додає їх до дерева, якщо вони не утворюють цикл.

Обидва алгоритми базуються на жадібному підході, який полягає у виборі локально оптимального рішення на кожному кроці з надією отримати глобально оптимальний результат. Жадібні алгоритми є ефективними для широкого класу задач, але їх застосування можливе лише за певних умов, коли локальний вибір гарантує глобальну оптимальність.

Складність алгоритмів побудови мінімального остовного дерева також залежить від реалізації. Наприклад, алгоритм Крускала з використанням структури неперетинних множин має складність $\Theta(E \log E)$, що є ефективним для більшості практичних задач.

Жадібні алгоритми використовуються не лише для задач на графах, але й у багатьох інших областях, зокрема у задачах оптимізації, планування та розподілу ресурсів. Їх основною перевагою є простота реалізації та висока швидкодія.

Роль графових структур у задачах програмної інженерії є надзвичайно важливою. Вони використовуються для моделювання залежностей між компонентами програм, аналізу потоків даних, оптимізації виконання завдань та побудови складних систем. Наприклад, графи застосовуються у компіляторах для побудови графів залежностей, у мережевих системах – для маршрутизації, у базах даних – для аналізу зв'язків між даними.

Особливе значення мають графи у задачах оптимізації. Наприклад, у логістиці вони використовуються для визначення оптимальних маршрутів доставки, у телекомунікаціях – для побудови ефективних мереж, у системах управління – для планування виконання завдань.

Важливим аспектом є також масштабованість графових алгоритмів. У сучасних системах обсяги даних можуть бути дуже великими, тому алгоритми повинні забезпечувати ефективну обробку навіть при значній кількості вершин і ребер. Це вимагає використання оптимальних структур даних і алгоритмічних підходів.

Таким чином, алгоритми на графах є потужним інструментом для розв'язування складних задач у програмній інженерії. Пошук найкоротших шляхів, побудова мінімальних остовних дерев і застосування жадібних алгоритмів дозволяють ефективно аналізувати та оптимізувати системи. Розуміння цих алгоритмів і їх властивостей є необхідною умовою для створення сучасних програмних продуктів із високою продуктивністю та надійністю.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

Основна

1. Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford. Introduction to Algorithms. 4th ed. Cambridge, MA: MIT Press, 2022. 1312 p. ISBN 9780262046305.
2. Skiena, Steven S. The Algorithm Design Manual. 3rd ed. Cham: Springer, 2020. 804 p. ISBN: 9783030542566, 3030542564.
3. Sedgewick, Robert, Wayne, Kevin. Algorithms. 4th ed. Boston: Addison-Wesley, 2011. 976 p. ISBN 9780321573513.
4. Goodrich, Michael T., Tamassia, Roberto, Goldwasser, Michael H. Data Structures and Algorithms in Python. Hoboken, NJ: Wiley, 2013. 768 p. ISBN 9781118290279.
5. Erickson, Jeff. Algorithms. 2023. (open textbook, University of Illinois). URL: https://jeffe.cs.illinois.edu/teaching/algorithms/

Допоміжна

6. Weiss, Mark Allen. Data Structures and Algorithm Analysis in C++. 4th ed. Boston: Pearson, 2014. 635 p. ISBN 9780132847377.
7. Laakmann McDowell, Gayle. Cracking the Coding Interview. 6th ed. Palo Alto: CareerCup, 2015. 706 p. ISBN: 9780984782864, 0984782869.
8. Kleinberg, J., Tardos, É. (2013). Algorithm Design. Pearson. 2013. 823p. ISBN: 9781292023946, 1292023945.
9. Chen, H. (2023). Computability and Complexity. MIT Press. 2023. 416p. ISBN: 9780262048620, 0262048620.
10. Мелешко Є. В., Якименко М. С., Поліщук Л. І. Алгоритми та структури даних: навчальний посібник. Кропивницький: Видавець Лисенко В. Ф., 2019. 156 с.

Інформаційні ресурси

11. Programiz. URL: https://www.programiz.com/dsa
12. GeeksforGeeks. URL: https://www.geeksforgeeks.org
13. VisuAlgo. URL: https://visualgo.net
14. CP-Algorithms. URL: https://cp-algorithms.com

Навчальне видання

Мірошник Марина Анатоліївна

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Опорний конспект лекцій для здобувачів вищої освіти,
які навчаються за спеціальностями галузі «Інформаційні технології»

Українською мовою