ДЕРЖАВНА СЛУЖБА СТАТИСТИКИ УКРАЇНИ
НАЦІОНАЛЬНА АКАДЕМІЯ СТАТИСТИКИ, ОБЛІКУ ТА АУДИТУ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ НАСОА
УКРАЇНСЬКИЙ МОВНО-ІНФОРМАЦІЙНИЙ ФОНД НАН УКРАЇНИ
КОМПАНІЯ SIGMA SOFTWARE
АТ «ІДЕЯ БАНК»

ЦИФРОВІЗАЦІЯ СУСПІЛЬСТВА ТА БІЗНЕСУ: ВИКЛИКИ ТА МОЖЛИВОСТІ ДЛЯ МОЛОДІ

МАТЕРІАЛИ
VI ВСЕУКРАЇНСЬКОЇ СТУДЕНТСЬКОЇ НАУКОВОЇ КОНФЕРЕНЦІЇ



Київ – 2026

Державна служба статистики України
Національна академія статистики, обліку та аудиту
Кафедра інформаційних технологій НАСОА
Український мовно-інформаційний фонд НАН України
Компанія Sigma Software
АТ «Ідея Банк»

ЦИФРОВІЗАЦІЯ СУСПІЛЬСТВА ТА БІЗНЕСУ: ВИКЛИКИ ТА МОЖЛИВОСТІ ДЛЯ МОЛОДІ

VI Всеукраїнська студентська наукова конференція

Київ – Інтерсервіс – 2026

УДК: 004.032.8:[316.3+334.72]:005.334–053.81
Ц 75

Цифровізація суспільства та бізнесу : виклики та можливості для молоді : матеріали VI Всеукраїнської студентської наукової конференції пам’яті доктора економічних наук, професора Кадієвського Володимира Андрійовича. Київ : Інтерсервіс. 2026. 68 с.

ISBN 978-966-999-638-1

Матеріали VI Всеукраїнської студентської наукової конференції містять тези доповідей, повідомлень та виступів її учасників, у яких викладено наукові результати аналітичного осмислення сутності сучасних проблем цифровізації суспільства та бізнесу, та можливостей для молоді у їх розв’язанні. Для докторантів, аспірантів та студентів економічних та соціально-гуманітарних спеціальностей.

Рекомендовано до друку Вченою радою
Національної академії статистики, обліку та аудиту
(протокол № 15 від 26 червня 2026)

Адреса оргкомітету конференції:
Україна, 04107, м. Київ,
вул. Підгірна, 1;
тел.: (044) 486–31–17; (067) 763–86–08
e-mail: nasoaconference@gmail.com

Відповідальність за зміст представлених матеріалів несуть автори. Редакційна колегія залишає за собою право на їх літературне редагування та відмову від друкування тез у разі невідповідності оформлення.

ISBN 978-966-999-638-1

© Національна академія статистики, обліку та аудиту, 2026
© National Academy of Statistics, Accounting and Audit, 2026

Редакційна колегія

Голова редакційної колегії:

Пилипенко О. І., декан обліково-статистичного факультету Національної академії статистики, обліку та аудиту, доктор економічних наук, доцент.

Заступник голови:

Ставицький О. В., завідувач кафедри інформаційних технологій Національної академії статистики, обліку та аудиту, кандидат економічних наук, доцент;

Члени редакційної колегії:

Надутенко М. В., старший науковий співробітник Українського мовно-інформаційного фонду НАН України, керівник Загальноукраїнського центру словникарства, кандидат філологічних наук;

Надутенко М. В., завідувач відділу інформатики УМІФ НАНУ, кандидат технічних наук.

Янцов В. В., L&D Program Manager, компанія Sigma Software;

Тарарук К. Ю., Head of Recruitment, компанія АТ «Ідея Банк» .

Учений секретар конференції:

Савченко С. С., старший викладач кафедри інформаційних технологій Національної академії статистики, обліку та аудиту.

ЗМІСТ

Ахметова А. О.
Еволюція UX-дизайну в соціальних мережах від фото до AI-контенту……..….8

Білошапка О. М.
Асоціативний масив як абстрактний тип даних:
реалізація на основі хеш-таблиці в Java……..…...10

Булатовський Ю. В.
Черга в обчислювальних системах:
теоретичні основи дисципліни FIFO та алгоритмічне застосування……...…….13

Голубенко М. С.
Json Web Token як механізм авторизації:
структура, підпис і застосування в веб-системах…….....15

Джапаєва А. Е.
Граф як структура даних: представлення та алгоритми обходу……...17

Коваленко І. Р.
Рекурсія та динамічне програмування:
методи розробки алгоритмів та оптимізація……..…….....20

Ковальова І. О.
Цифровізація економіки та її вплив на сучасне суспільство…….……22

Корольова А. А.
Бітовий вектор як компактна структура даних:
побітові операції та застосування для представлення множин…….…….…...25

Мельник О. В.
Алгоритми сортування:
класифікація, аналіз складності та порівняльна характеристика…….27

Нємов О. С.
Червоно-чорне дерево:
самобалансування та гарантована складність операцій……30

Павлюк Є. П.
REST API як архітектурний стиль: принципи проектування та реалізація…….32

Палій В. О.
Алгоритми пошуку у великих наборах даних:
порівняльний аналіз бінарного, Фібоначчі та хеш-пошуку…….…….35

Пархоменко М. І.
Клас для обробки текстової інформації:
реалізація рядкових операцій засобами Java 37

Пархут М. В.
Індекси в реляційних базах даних:
структура, типи та вплив на продуктивність запитів…….....40

Рогульчик Т. С.
AVL-дерево як самобалансована структура даних:
алгоритми балансування та аналіз складності операцій……..……..42

Рубель М. С.
Дек як узагальнення стека і черги:
організація двосторонньої черги та її алгоритмічне застосування……..…….45

Садовнича А. І.
ІТ як інструмент самовираження та їх вплив на емоційний стан людини…..….47

Слободяник Б.
Об'єктно-орієнтоване програмування:
принципи інкапсуляції, наслідування та поліморфізму……..……..51

Ставицький І. О.
Бінарне дерево пошуку: структура, операції та аналіз складності……...53

Телетьон М. В.
Хешування та шифрування:
теоретичні основи та застосування в програмних системах…….……56

Ціцура П. В.
Вектор як стратегія розширення динамічного масиву
та аналіз складності операцій…….....58

Шевченко М. І.
Стек як основна структура даних:
принципи організації та області застосування…….…….....61

Шиловських Т. С.
Черга з пріоритетами та двійкова купа:
теоретичні основи і застосування в алгоритмах……63

Щербак П. Ю.
Використання хмарних технологій
у цифровій трансформації підприємств…….....66

ТЕЗИ ВИСТУПІВ

СТУДЕНТІВ-УЧАСНИКІВ КОНФЕРЕНЦІЇ

Ахметова А. О.,
здобувачка вищої освіти першого (бакалаврського) рівня,
спеціальність «Комп'ютерні науки»;
науковий керівник:
Ставицький О. В.,
кандидат економічних наук, доцент
завідувач кафедри інформаційних технологій,
Національна академія статистики,
обліку та аудиту

ЕВОЛЮЦІЯ UX–ДИЗАЙНУ В СОЦІАЛЬНИХ МЕРЕЖАХ ВІД ФОТО ДО AI-КОНТЕНТУ

Упродовж останніх двох десятиліть соціальні мережі пройшли стрімкий шлях трансформації – від простих платформ для обміну текстовими повідомленнями та фотографіями до складних екосистем, у яких домінують алгоритми, персоналізація та штучний інтелект. У цьому процесі UX-дизайн (User Experience Design) відіграє ключову роль, адже саме він визначає, як користувач взаємодіє з контентом, які емоції отримує та наскільки ефективно досягає своїх цілей.

На ранніх етапах розвитку соціальних мереж UX-дизайн був орієнтований на простоту та функціональність. Платформи на кшталт перших версій Facebook пропонували базовий інтерфейс із мінімальним набором функцій: профіль користувача, стрічка новин та можливість публікації текстів і зображень. Основний акцент робився на зручності навігації та швидкому доступі до інформації. Контент мав переважно статичний характер – фотографії, які користувачі завантажували вручну, без значної обробки чи автоматизації.

З розвитком мобільних технологій UX-дизайн почав змінюватися у напрямку візуалізації та інтерактивності. Поява таких платформ, як Instagram, спричинила зміщення фокусу на візуальний контент. Інтерфейси стали більш мінімалістичними, з акцентом на великі зображення, плавні анімації та інтуїтивні жести. UX-дизайнери почали враховувати поведінкові патерни користувачів, зокрема звичку швидкого скролінгу та короткої концентрації уваги. Це призвело до появи нових форматів – stories, reels, коротких відео – які оптимізовані під швидке споживання контенту.

Наступний етап еволюції UX-дизайну пов'язаний із впровадженням алгоритмічних стрічок. Соціальні мережі почали використовувати великі обсяги даних для персоналізації контенту, що значно змінило користувацький досвід. Стрічка новин перестала бути хронологічною і перетворилася на індивідуалізований простір, сформований на основі інтересів, поведінки та взаємодій користувача. Це означало, що UX-дизайн більше не обмежується

лише інтерфейсом – він включає також логіку роботи алгоритмів і способи подачі контенту.

Сьогодні ж ми спостерігаємо новий етап – інтеграцію штучного інтелекту у створення та споживання контенту. Як зазначає Ситник О., штучний інтелект стає все більш актуальним у медіаіндустрії, оскільки він пропонує значні переваги, зокрема підвищену ефективність, персоналізацію, аналіз даних, створення контенту та контроль якості. Оскільки технологія штучного інтелекту продовжує розвиватися, вона, ймовірно, матиме ще більший вплив на медіаіндустрію та спосіб споживання та взаємодії з медіаконтентом [5, с. 263].

Інтеграція ШІ суттєво змінює підходи до UX-дизайну. По-перше, зростає рівень персоналізації: системи можуть адаптувати інтерфейс, контент і навіть функціонал під конкретного користувача. По-друге, змінюється сам процес створення контенту. За даними Felix Rose–Collins, ШІ на основі комп'ютерного зору відіграє ключову роль у створенні зображень і відео; він дає змогу автоматично розпізнавати об'єкти, додавати теги, покращувати візуальні елементи, сприяючи створенню професійної графіки для вебсайтів і соціальних мереж [2]. Це означає, що UX-дизайн має враховувати не лише споживання, а й генерацію контенту користувачем за допомогою інтелектуальних інструментів.

Крім того, штучний інтелект змінює роль користувача у соціальних мережах. Якщо раніше він був переважно споживачем або творцем контенту, то тепер стає співтворцем разом із алгоритмами. Наприклад, генеративні моделі можуть пропонувати ідеї постів, автоматично редагувати фото або навіть створювати відео. У цьому контексті UX-дизайн має забезпечити прозору, зрозумілу та етичну взаємодію між людиною та ШІ.

Важливим аспектом є також професійне використання ШІ в креативних індустріях. У дослідженні освітньої лабораторії «Projector AI Lab» зазначено, що штучний інтелект найбільше використовують дизайнери, копірайтери, представники маркетингу та PR [3]. Це свідчить про те, що UX-дизайн стає міждисциплінарною сферою, де поєднуються технології, творчість і аналітика. Дизайнери вже не лише створюють інтерфейси, а й працюють із даними, алгоритмами та інструментами автоматизації.

Отже, еволюція UX-дизайну в соціальних мережах відображає загальні тенденції розвитку цифрових технологій – від простоти до складності, від статичності до динамічності, від ручного створення контенту до автоматизованого. Перехід від фото до AI-контенту не лише змінює форму взаємодії, а й переосмислює роль користувача, дизайнера та самої платформи. У майбутньому можна очікувати ще глибшої інтеграції штучного інтелекту, що зробить UX-дизайн ще більш адаптивним, персоналізованим і водночас відповідальним.

Список використаних джерел

1. Ситник О. Проблематика впровадження штучного інтелекту в сучасних ЗМІ та медіатехнологіях // Український інформаційний простір. 2023. № 2 (12). С. 252–265.
2. Rose–Collins F. Створення контенту зі штучним інтелектом: підвищення продуктивності та креативності [Електронний ресурс] // Ranktracker. 20 травня 2024. Режим доступу: https://www.ranktracker.com/uk/blog/ai–powered–content–creation–boosting–productivity–and–creativity/ (дата звернення: 03.05.2026).
3. Дослідження використання штучного інтелекту в українських компаніях [Електронний ресурс] // Projector AI Lab. Березень 2023. Режим доступу: https://prjctr.com/library/video/shtuchnij–intelekt–ta–dizajn–doslidzhennya (дата звернення: 03.05.2026).

Білошапка О. М.,
студент 3 курсу ОПС «Фаховий молодший бакалавр»
спеціальності 121 «Інженерія програмного забезпечення»
освітньо-професійної програми «Інженерія програмного забезпечення»;
науковий керівник:
Байдінов Д. О.,
старший викладач,
голова циклової комісії з програмування
Фаховий коледж бізнесу та аналітики НАСОА

АСОЦІАТИВНИЙ МАСИВ ЯК АБСТРАКТНИЙ ТИП ДАНИХ: РЕАЛІЗАЦІЯ НА ОСНОВІ ХЕШ-ТАБЛИЦІ В JAVA

Більшість програмних систем працюють не лише зі значеннями, але й з відображеннями: ім'я користувача на профіль, URL на ресурс, код продукту на ціну. Абстракція зберігання даних у формі пар ключ–значення є настільки природною для комп'ютерних завдань, що більшість сучасних мов програмування включають асоціативний масив як вбудований тип. Внутрішня реалізація цієї абстракції через хеш-таблицю є важливою для свідомого використання стандартних колекцій та для створення власних структур даних.

Асоціативний масив (словник, карта) є абстрактним типом даних, який підтримує три основні операції. Операція put(key, value) додає пару або оновлює значення для існуючого ключа. Операція get(key) повертає значення для ключа або null, якщо ключ відсутній. Операція remove(key) видаляє пару за ключем. Допоміжні операції containsKey, size та isEmpty доповнюють

інтерфейс. Ключ у межах одного словника є унікальним: повторна вставка ключа перезаписує попереднє значення [1].

Найефективнішою реалізацією асоціативного масиву є хеш-таблиця. Хеш-функція відображає ключ на ціле число і обчислює індекс у масиві бакетів за формулою index = hash(key) % capacity. При якісній хеш-функції ключі розподіляються рівномірно по масиву, і кожен бакет у середньому містить n/capacity елементів. Доступ до бакету за індексом виконується за O(1), тому середня складність put, get і remove також становить O(1) [2].

Фундаментальною проблемою хеш–таблиць є колізії: ситуації, коли різні ключі мають однаковий хеш-індекс. Метод ланцюжків вирішує колізії через збереження у кожному бакеті зв'язаного списку пар ключ–значення. При колізії нова пара додається до списку бакета, а пошук перебирає список до знаходження ключа. Метод відкритої адресації не використовує допоміжних структур: при колізії виконується пробування наступних бакетів за певною схемою до знаходження вільного місця. Метод ланцюжків стійкіший до перевантаження і простіший в реалізації [3].

Коефіцієнт заповнення (load factor) визначається як n / capacity, де n є поточною кількістю елементів. При перевищенні порогу (зазвичай 0,75) виконується рехешування: виділяється новий масив вдвічі більшого розміру, і всі елементи перерозподіляються за новими індексами. Рехешування займає O(n) часу, але амортизована вартість вставки залишається O(1). Підтримання низького коефіцієнта заповнення забезпечує коротші ланцюжки і стабільний O(1) доступ.

Клас CustomMap <K, V> реалізує асоціативний масив з параметризованими типами ключа і значення. Внутрішній клас Node <K, V> зберігає пару ключ-значення і посилання на наступний вузол для реалізації ланцюжка. Масив Node [] buckets є основним сховищем, початкова ємність встановлюється рівною 16. Метод hashCode ключа визначає бакет, де операція взяття модуля за ємністю гарантує коректний індекс. При коефіцієнті заповнення понад 0,75 ємність подвоюється і виконується рехешування.

У Java стандартна реалізація асоціативного масиву представлена класом java.util.HashMap<K, V>. HashMap використовує масив бакетів і метод ланцюжків для вирішення колізій, починаючи з Java 8 ланцюжки автоматично перетворюються на збалансовані дерева при довжині більше 8 елементів. LinkedHashMap зберігає порядок вставки через додатковий двозв'язний список вузлів. TreeMap реалізує відсортований словник через червоно-чорне дерево з O(log n) для всіх операцій. Вибір між ними визначається вимогою до порядку і необхідністю гарантованої складності.

Якість хеш-функції є одним з ключових факторів продуктивності хеш-таблиці. Хороша хеш-функція має дві властивості: рівномірний розподіл по всьому діапазону індексів та детермінованість: однаковий вхід завжди дає однаковий результат. У Java метод hashCode визначений у класі Object і може

бути перевизначений у користувацьких класах. Контракт hashCode вимагає, що якщо два об'єкти рівні за equals, їх hashCode мусить бути однаковим. Порушення цього контракту при використанні таких об'єктів як ключів у HashMap призводить до некоректної роботи пошуку: елемент буде знайти неможливо, хоча він фізично присутній у структурі.

Асоціативний масив є однією з найфундаментальніших структур даних у практичному програмуванні. Реалізація CustomMap<K, V> на основі хеш-таблиці з методом ланцюжків демонструє, як проста ідея хешування перетворює потенційно O(n) пошук у середньоефективний O(1). Розуміння механізмів хеш-функції, рехешування, вирішення колізій і контракту equals/hashCode є необхідною основою для ефективного використання стандартних колекцій Java у промисловій розробці і проектування власних структур зберігання даних.

Список використаних джерел

1. Марченко О. І., Марченко О. О. Структури даних та алгоритми : підручник. Ч. 1. Київ : Просвіта, 2024. 268 с.
2. Васильєв О. М. Програмування мовою Java. Київ : Богдан, 2019. 696 с.
3. Татарчук Д. Д., Діденко Ю. В., Свєчніков Г. С. Об'єктно-орієнтоване програмування мовою Java : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2024. 153 с.

Булатовський Ю. В.,
студент 3 курсу ОПС «Фаховий молодший бакалавр»
спеціальності 121 «Інженерія програмного забезпечення»
освітньо-професійної програми «Інженерія програмного забезпечення»;
науковий керівник:
Савченко С. С.,
старший викладач,
циклова комісія з програмування
Фаховий коледж бізнесу та аналітики НАСОА

ЧЕРГА В ОБЧИСЛЮВАЛЬНИХ СИСТЕМАХ: ТЕОРЕТИЧНІ ОСНОВИ ДИСЦИПЛІНИ FIFO ТА АЛГОРИТМІЧНЕ ЗАСТОСУВАННЯ

Математична теорія черг виникла на початку XX століття у роботах данського інженера Агнера Ерланга, який вивчав навантаження на телефонні станції. Ерланг сформулював основні закономірності масового обслуговування задовго до винайдення комп'ютерів, але його модель виявилась настільки точним описом реальності, що її перейняли комп'ютерні науки без суттєвих змін. Сьогодні черга як структура даних є прямою програмною реалізацією цієї математичної абстракції.

Черга реалізує принцип FIFO (перший прийшов, перший вийшов): елемент, який першим входить у структуру, першим з неї видаляється. Це забезпечує хронологічний порядок обробки елементів і запобігає «голодуванню», коли певний елемент чекає на обслуговування нескінченно. Саме це відрізняє чергу від стека і робить її фундаментальним інструментом у будь-якій ситуації, де справедливість порядку є важливою [1].

Найелегантнішою реалізацією черги на масиві є кільцевий буфер. Два цілі індекси, head і tail, вказують на початок і кінець черги і рухаються за формулою index = (index + 1) % capacity. Коли tail досягає кінця масиву, він повертається на початок і використовує вже звільнені комірки. Така організація забезпечує O(1) для enqueue і dequeue без будь-якого переміщення даних і є стандартом у реалізаціях черг в операційних системах [2].

Апаратна і системна підтримка черг присутня у всій архітектурі сучасних комп'ютерів. Буфер клавіатури зберігає натискання клавіш у черзі для читання процесом. Спулер друку організовує завдання у чергу і відправляє їх на принтер по одному. Планувальник дискового I/O обслуговує запити до секторів диска, розподіляючи їх у черги для мінімізації переміщення головки. У всіх цих випадках черга вирішує проблему невідповідності швидкості між виробником і споживачем даних.

Найважливішим алгоритмічним застосуванням черги є пошук у ширину (BFS) графа. Алгоритм починає з першої вершини, поміщає її в чергу, а далі на

кожному кроці видаляє вершину з голови, відвідує її і додає всіх невідвіданих сусідів у хвіст. Черга забезпечує обробку вершин рівно в порядку зростання відстані від початкової точки, що робить BFS оптимальним для знаходження найкоротшого шляху в незважених графах [3].

Шаблон виробник–споживач є прикладом застосування черги у багатопотоковому програмуванні. Один або кілька потоків генерують дані і розміщують їх у спільній черзі, а інші потоки видаляють і обробляють елементи незалежно. Черга діє як буфер, що розділяє виробника і споживача за швидкістю і дозволяє їм рухатися з різною швидкістю без синхронізації один з одним.

У Java ієрархія черг розгалужується залежно від поведінкових потреб. Базовий інтерфейс Queue розширюється Deque для двосторонньої черги і BlockingQueue для потокобезпечної версії. ArrayDeque реалізує кільцевий буфер і є найкращим вибором для однопотокових задач. ArrayBlockingQueue і LinkedBlockingQueue мають механізм блокування, коли черга порожня або повна, що повністю автоматизує синхронізацію у шаблоні виробник-споживач.

Черга з пріоритетами є природним розширенням базової черги, де порядок видалення визначається не часом надходження, а числовим пріоритетом елемента. Реалізація через купу забезпечує видалення мінімуму і додавання за O(log n). Черга з пріоритетами є основою алгоритму Дейкстри для знаходження найкоротшого шляху і алгоритму Прима для побудови мінімального остовного дерева.

Аналіз часової складності операцій черги показує, що для обох реалізацій, масивної і зв'язної, основні операції enqueue і dequeue виконуються за O(1). Просторова складність становить O(n), де n є поточна кількість елементів. Кільцевий буфер фіксованого розміру не потребує виділення пам'яті для операцій, тому він підходить для систем реального часу з жорсткими вимогами до затримки.

Черга є особливим випадком лінійних структур даних, оскільки має сильне теоретичне підґрунтя та широке практичне застосування. Від математичної моделі Ерланга до кільцевого буфера в ядрі операційної системи, до алгоритму BFS та блокуючих черг у багатопоточному коді, ця структура є актуальною для кожного рівня програмного стеку. Черга та її варіації є необхідними для знання в системному програмуванні та алгоритмах графів.

Список використаних джерел

1. Скулиш М. А., Суліма С. В. Алгоритми та структури даних : лабораторний практикум. Київ : КПІ ім. Ігоря Сікорського, 2021. 109 с.

2. Васильєв О. М. Програмування мовою Java. Київ : Богдан, 2019. 696 с.

3. Труніна Г. О., Настенко Д. В., Нестерко А. Б. Обчислювальна техніка та програмування : конспект лекцій. Ч. 1. Київ : КПІ ім. Ігоря Сікорського, 2020. 117 с.

Голубенко М. С.,
студент 3 курсу ОПС «Фаховий молодший бакалавр»
спеціальності 121 «Інженерія програмного забезпечення»
освітньо-професійної програми «Інженерія програмного забезпечення»;
науковий керівник:
Байдінов Д. О.,
старший викладач,
голова циклової комісії з програмування
Фаховий коледж бізнесу та аналітики НАСОА

JSON WEB TOKEN ЯК МЕХАНІЗМ АВТОРИЗАЦІЇ: СТРУКТУРА, ПІДПИС І ЗАСТОСУВАННЯ В ВЕБ–СИСТЕМАХ

Стандартний підхід до авторизації у веб–додатках базується на серверних сесіях: після входу в систему сесія записується на сервері, і ідентифікатор відправляється клієнту. При масштабуванні на більше серверів нам потрібен доступ до спільного сховища сесій, і це може бути складно спроектувати простим способом. JSON Web Token є іншою альтернативою: контекст авторизації може бути відправлений безпосередньо клієнту, і сервер не відповідає за будь-який стан.

JWT складається з трьох частин, розділених крапками: заголовок, корисне навантаження і підпис. Кожна частина є рядком у форматі Base64URL. Заголовок вказує тип токена і алгоритм підпису. Payload містить твердження (claims) про суб'єкт – це довільний JSON–об'єкт з інформацією про користувача і дозволи. Підпис обчислюється на основі заголовка і payload і запобігає підробці [1].

Стандарт RFC 7519 визначає набір зареєстрованих claims. iss (issuer) ідентифікує сервер, що видав токен. sub (subject) ідентифікує суб'єкт, зазвичай ідентифікатор користувача. exp (expiration time) містить час закінчення дії токена у форматі Unix timestamp. iat (issued at) фіксує момент видачі. nbf (not before) вказує, раніше якого моменту токен недійсний. Перевірка exp є обов'язковою на сервері: прийняття токена без перевірки терміну дії є критичною вразливістю.

Підпис токена може бути симетричним або асиметричним. Алгоритм HS256 (HMAC SHA–256) використовує спільний секретний ключ: обидві

сторони знають ключ і можуть перевірити підпис. Це зручно, але потребує безпечного поширення секрету між сервісами. Алгоритм RS256 (RSA SHA–256) використовує пару ключів: сервер авторизації підписує приватним ключем, а ресурсний сервер перевіряє публічним – який можна публічно розповсюджувати без ризику підробки [2].

Типовий сценарій використання JWT такий: клієнт надсилає логін і пароль, сервер автентифікує їх і видає підписаний JWT. З кожним наступним запитом клієнт додає токен до заголовка Authorization зі схемою Bearer. Сервер перевіряє підпис і заяви, і якщо токен є дійсним, він обробляє запит без доступу до бази даних або сховища сесій.

JWT має кілька відомих вразливостей, яких слід уникати. Атака "none" алгоритму замінює алгоритм підпису на none: деякі бібліотеки приймали непідписані токени. Слабкий секретний ключ робить HS256 вразливим до brute–force. Відсутність перевірки exp дозволяє використовувати прострочені токени. Зберігання JWT у localStorage робить його вразливим до XSS–атак; httpOnly cookies є безпечнішим варіантом [3].

Оскільки JWT не можна відкликати до закінчення строку дії, access token зазвичай має короткий термін (15 хвилин). Разом з цим видається токен оновлення – довготривалий токен, який зберігається в репозиторії на сервері і може бути використаний для отримання нового токена доступу без повторного входу. Якщо сервер скомпрометовано, токен оновлення додається до чорного списку або видаляється з репозиторію.

У Java ми можемо працювати з JWT за допомогою бібліотек, таких як jjwt (io.jsonwebtoken) або Nimbus JOSE + JWT. jjwt має зручний API: Jwts.builder() створює токен, і він підписується за допомогою signWith(key, algorithm) для перевірки, потім Jwts.parserBuilder().setSigningKey(key).build().parseClaimsJws(token) перевіряє його. У Spring Security фільтр JWT перехоплює запити, перевіряє токен і встановлює SecurityContext.

JWT часто використовується в OAuth 2.0, делегованому протоколі авторизації. У цій схемі сервер авторизації видає токен доступу у форматі JWT після того, як користувач надає дозвіл додатку діяти від його імені. Сервер ресурсів перевіряє підпис токена без звернення до сервера авторизації, що дозволяє системі масштабуватися горизонтально. OpenID Connect надбудовується над OAuth 2.0 і додає ID token з інформацією про автентифікованого користувача [1].

Порівняно з непрозорими (opaque) токенами, JWT має як переваги, так і обмеження. Перевага – самодостатність: сервер не потребує звернення до бази для перевірки токена. Обмеження – неможливість миттєвого відкликання: навіть якщо користувач виходить, токен залишається дійсним до закінчення exp. Для чутливих операцій (зміна пароля, фінансові транзакції) це потребує або дуже короткого терміну дії, або дод. перевірки через token blacklist.

JSON Web Token є компактним і самодостатнім механізмом передачі авторизаційного контексту між сервісами без збереження стану на сервері. Його структура, стандартизований набір claims і підтримка як симетричного, так і асиметричного підпису роблять JWT стандартом де-факто для авторизації в мікросервісних архітектурах і REST API.

Список використаних джерел

1. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT) : RFC 7519. Internet Engineering Task Force, 2015. 30 p.
2. Шилдт Г. Java. Повне керівництво. 10–те вид. Київ : Діалектика, 2019. 1376 с.
3. Татарчук Д. Д., Діденко Ю. В., Свєчніков Г. С. Об'єктно–орієнтоване програмування мовою Java : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2024. 153 с.

Джапаєва А. Е.,
студентка 2 курсу ОПС «Фаховий молодший бакалавр»
спеціальності 121 «Інженерія програмного забезпечення»
освітньо-професійної програми «Інженерія програмного забезпечення»;
науковий керівник:
Байдінов Д. О.,
старший викладач,
голова циклової комісії з програмування
Фаховий коледж бізнесу та аналітики НАСОА

ГРАФ ЯК СТРУКТУРА ДАНИХ: ПРЕДСТАВЛЕННЯ ТА АЛГОРИТМИ ОБХОДУ

Дерева описують ієрархічні відношення між об'єктами, але реальний світ рясніє зв'язками іншого характеру: дороги між містами, гіперпосилання між сторінками, підписки між акаунтами у соціальних мережах, залежності між модулями програмного проекту. Граф є математичною абстракцією, що узагальнює всі ці структури: він складається з вершин і ребер між ними, не накладаючи ніяких обмежень на форму з'єднань [1].

Граф G визначається як пара (V, E), де V – множина вершин, E – множина ребер. Орієнтований граф (діграф) має спрямовані ребра: ребро (u, v) означає зв'язок від u до v, але не навпаки. Неорієнтований граф має двонаправлені ребра. Зважений граф присвоює кожному ребру числову вагу – наприклад,

відстань, вартість або пропускну здатність. Граф без циклів, де з будь-якої вершини досяжна будь-яка інша, є деревом [2].

Матриця суміжності є двовимірним масивом розміру |V|×|V|, де елемент [i][j] дорівнює вазі ребра між вершинами і і j або нулю/нескінченності, якщо ребра немає. Перевірка наявності ребра між двома вершинами виконується за O(1). Недолік: просторова складність $O(V^2)$ є неприйнятною для розріджених графів, де кількість ребер значно менша за V^2.

Список суміжності зберігає для кожної вершини список її сусідів. Просторова складність O(V + E) робить список суміжності ефективним для розріджених графів. Перевірка наявності ребра має складність O(deg(v)), де deg(v) – це кількість сусідів вершини v. Список суміжності реалізується в Java у вигляді Map<Integer, List<Integer>> або Map<Integer, Set<Integer>>.

BFS проходить граф шар за шаром: спочатку всі сусіди джерела, потім їхні сусіди і так далі. Єдина необхідна структура тут – черга: беремо вершину з голови і додаємо невідвіданих сусідів у хвіст. Саме через цей порядок BFS гарантує знаходження найкоротшого шляху в невагових графах за O(V + E).

DFS спускається однією гілкою до кінця – і лише коли далі нікуди, повертається й пробує наступну. Рекурсивна реалізація мінімальна: відвідати вершину, позначити її, перейти до сусідів. Для великих графів, де глибина стека може стати проблемою, використовують ітеративну версію з явним стеком. DFS застосовується для виявлення циклів, топологічного сортування і знаходження зв'язних компонент, також за O(V + E) [3].

Алгоритм Дейкстри знаходить найкоротші шляхи від джерела до всіх вершин у зваженому графі з невід'ємними вагами. Він використовує чергу з пріоритетами, де пріоритет вершини – поточна найкоротша відома відстань. Топологічне сортування впорядковує вершини орієнтованого ациклічного графа так, що всі ребра спрямовані від раніших до пізніших. Алгоритм Прима будує мінімальне остовне дерево, жадібно додаючи ребра мінімальної ваги.

Алгоритм Беллмана-Форда знаходить найкоротші шляхи від джерела у графі з від'ємними вагами ребер. Він виконує V-1 релаксацій усіх ребер, а потім перевіряє наявність від'ємних циклів: якщо ще одна релаксація змінює відстань – цикл існує. Складність O(V·E) робить його повільнішим за Дейкстру, але це єдиний алгоритм, який коректно працює з від'ємними вагами, і тому він використовується в протоколах маршрутизації, зокрема RIP.

У стандартній бібліотеці Java немає вбудованого класу графа. Граф реалізують вручну або використовують бібліотеки на зразок JGraphT, що надає реалізації SimpleGraph, DirectedGraph і DefaultWeightedEdge. Представлення через Map<Integer, List<Integer>> є типовим вибором для конкурсних задач: воно просте, достатньо ефективне і не потребує зовнішніх залежностей.

Алгоритм Флойда-Воршала знаходить найкоротші шляхи між усіма парами вершин за $O(V^3)$ і $O(V^2)$ пам'яті. Він використовує матрицю відстаней dist[i][j] і послідовно розглядає кожну вершину як проміжну. На відміну від

Дейкстри, цей алгоритм коректно обробляє ребра від'ємної ваги (за відсутності від'ємних циклів) і підходить для густих графів із невеликою кількістю вершин.

Задача виявлення циклів у орієнтованому графі розв'язується DFS з відстеженням "сірих" вершин – тих, що перебувають у стеку рекурсії. Якщо під час обходу зустрічається сіра вершина, граф містить цикл. Це має практичне значення для аналізу залежностей у збірці проєктів: Maven і Gradle виконують саме такий аналіз перед побудовою графа задач компіляції.

Граф як структура даних є найзагальнішою з усіх розглянутих структур: дерево, список, черга – все це часткові випадки або варіації графа. Вибір між матрицею і списком суміжності визначається щільністю графа і профілем операцій. Алгоритми BFS і DFS є фундаментом, на якому будуються десятки спеціалізованих алгоритмів для задач маршрутизації, планування і аналізу мереж.

Список використаних джерел

1. Грудзинський Ю. Є. Алгоритми та структури даних : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2022. 215 с.
2. Бугаєва Л. М., Ковалюк Д. О. Алгоритми та структури даних : комп'ютерний практикум. Київ : КПІ ім. Ігоря Сікорського, 2022. 34 с.
3. Кузьменко І. М., Дацюк О. А. Базові алгоритми та структури даних : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2022. 137 с.

Коваленко І. Р.,
студент 4 курсу ОПС «Фаховий молодший бакалавр»
спеціальності 121 «Інженерія програмного забезпечення»
освітньо-професійної програми «Інженерія програмного забезпечення»;
науковий керівник:
Пантєєв Р. Л.,
кандидат технічних наук,
доцент кафедри інформаційних технологій,
Національна академія статистики, обліку та аудиту

РЕКУРСІЯ ТА ДИНАМІЧНЕ ПРОГРАМУВАННЯ: МЕТОДИ РОЗРОБКИ АЛГОРИТМІВ ТА ОПТИМІЗАЦІЯ

Деякі задачі за своєю природою визначаються через самих себе. Факторіал n це n, помножений на факторіал (n–1). Довжина дерева – одиниця плюс максимум довжин піддерев. Рекурсія – це техніка, що дозволяє функції викликати саму себе, і вона є природним способом вирішення задач з самоподібною або вкладеною структурою [1].

Кожна рекурсивна функція потребує двох речей: базового випадку (base case), що зупиняє рекурсію, і рекурсивного випадку (recursive case), що зводить задачу до меншого підзавдання. Без базового випадку функція викликає саму себе нескінченно, поки стек викликів не переповниться. У Java це називається StackOverflowError, коли глибина рекурсії перевищує ліміт потоку [2].

Кожен виклик функції додає новий кадр у стек викликів з локальними змінними та адресою повернення. Для рекурсії глибиною n це означає n одночасних кадрів. За замовчуванням Java дозволяє кілька тисяч рівнів рекурсії. На відміну від деяких функціональних мов, JVM не оптимізує хвостову рекурсію. Глибока рекурсія в Java зазвичай переписується в ітеративну форму з явним стеком [3].

Наївна рекурсивна реалізація числа Фібоначчі обчислює fib(n) = fib(n–1) + fib(n–2). Це призводить до дерева рекурсивних викликів, де fib(2) обчислюється окремо при кожному виклику – кількість викликів зростає експоненційно: O(2^n). Проблема в тому, що ті самі підзадачі вирішуються багато разів. Це явище називається перекриванням підзадач і є сигналом для застосування динамічного програмування.

Мемоізація (top–down DP) – це та сама рекурсія, але з кешем. Перед кожним обчисленням перевіряємо: а чи вже рахували це значення? Якщо так, повертаємося з кешу без рекурсії. Для Фібоначчі це рятує від O(2^n), оскільки кожне унікальне значення обчислюється лише один раз, і складність становить O(n). У Java зручно використовувати Map.computeIfAbsent для цього.

Табуляція (динамічне програмування знизу вгору) йде в протилежному напрямку – від базових випадків до цільового значення, заповнюючи таблицю

ітеративно без будь-якої рекурсії. У випадку Фібоначчі достатньо мати масив dp[], де dp[i] = dp[i–1] + dp[i–2]. Перевага табуляції – відсутність накладних витрат стека викликів і можливість оптимізувати пам'ять, зберігаючи лише кілька останніх значень замість всієї таблиці.

Динамічне програмування застосовується тоді, коли задача має дві властивості: перекриваючі підзадачі (ті самі підзадачі вирішуються багаторазово) і оптимальна підструктура (оптимальне рішення задачі будується з оптимальних рішень підзадач). Класичні приклади: найбільша спільна підпослідовність (LCS), задача про рюкзак, задача про видачу решти монетами. Для кожної з них наївна рекурсія є експоненціальною, а DP – поліноміальною.

Задача про найбільшу спільну підпослідовність двох рядків довжиною m і n вирішується через таблицю dp[m+1][n+1]. dp[i][j] зберігає довжину LCS перших і символів першого рядка і перших j символів другого. Якщо символи рівні, dp[i][j] = dp[i–1][j–1] + 1; інакше dp[i][j] = max(dp[i–1][j], dp[i][j–1]). Складність O(m*n) часу і пам'яті проти O(2^(m+n)) для наївної рекурсії.

Задача про розбиття числа N на суму натуральних доданків є класичним прикладом ДП на розбиттях: dp[i][j] – кількість способів подати j як суму доданків, кожен з яких не перевищує i. Вона застосовується у криптографії для оцінки ентропії ключових просторів і в теорії чисел. Обхід усіх варіантів розбиття наївно дає експоненційну складність, тоді як ДП виконується за $O(N^2)$.

Підхід "зустріч посередині" (meet–in–the–middle) дозволяє розв'язувати NP–важкі задачі за O(2^(n/2)) замість O(2^n). Для задачі про суму підмножини масив ділять навпіл: перебирають усі підмножини лівої і правої половин, для кожної суми лівої половини бінарним пошуком шукають відповідне доповнення у правій. Цей прийом поєднує рекурсію повного перебору з пошуком для скорочення вдвічі показника степеня.

Розділяй і володарюй (divide and conquer) – окрема рекурсивна парадигма, де задача ділиться на незалежні підзадачі. На відміну від ДП, підзадачі тут не перекриваються: MergeSort ділить масив навпіл, сортує кожну половину незалежно, а потім зливає результати. Швидкий пошук k–го порядкового елемента (QuickSelect), алгоритм Штрассена для множення матриць і швидке перетворення Фур'є – усі вони побудовані на цьому підході.

Рекурсія і динамічне програмування – не взаємовиключні підходи, а взаємодоповнюючі. Рекурсія дає природну декомпозицію задачі, мемоізація додає кешування без зміни структури коду, табуляція усуває рекурсію для максимальної ефективності. Вміння переходити між цими трьома формами і розпізнавати задачі з оптимальною підструктурою є ключовою навичкою при розробці ефективних алгоритмів.

Список використаних джерел

1. Марченко О. І., Марченко О. О. Структури даних та алгоритми : підручник. Ч. 1. Київ : Просвіта, 2024. 268 с.

2. Грудзинський Ю. Є. Алгоритми та структури даних : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2022. 215 с.
3. Васильєв О. М. Програмування мовою Java. Київ : Богдан, 2019. 696 с.

Ковальова І. О.,
здобувачка вищої освіти першого (бакалаврського) рівня,
спеціальність «Статистика цифрової економіки»;
науковий керівник:
Ставицький О. В.,
кандидат економічних наук, доцент,
завідувач кафедри інформаційних технологій,
Національна академія статистики, обліку та аудиту

ЦИФРОВІЗАЦІЯ ЕКОНОМІКИ ТА ЇЇ ВПЛИВ НА СУЧАСНЕ СУСПІЛЬСТВО

У сучасному світі технології розвиваються дуже швидко, і це безпосередньо впливає на всі сфери життя людей. Особливо це помітно в економіці, яка поступово переходить у цифровий формат. Сьогодні підприємства, організації та навіть державні установи активно використовують різні цифрові інструменти у своїй діяльності. Майже всі процеси – від виробництва до продажу товарів – пов’язані з використанням інтернету, програмного забезпечення та сучасних технологій. Саме тому цифровізація економіки є важливим етапом розвитку суспільства, адже вона дозволяє значно підвищити ефективність роботи та конкурентоспроможність країни [1].

На мою думку, однією з головних переваг цифровізації є те, що вона значно спрощує повсякденну діяльність підприємств. Якщо раніше багато операцій виконувалися вручну і займали багато часу, то зараз їх можна автоматизувати. Наприклад, існують спеціальні програми для обліку, аналізу та планування, які допомагають швидше обробляти інформацію. Це дозволяє не тільки економити час, а й зменшувати кількість помилок. Крім того, підприємства можуть швидше реагувати на зміни на ринку і приймати більш ефективні рішення.

Також важливою перевагою є розвиток електронної комерції. Сьогодні люди можуть купувати товари і послуги через інтернет, не виходячи з дому. Це дуже зручно, особливо в умовах швидкого темпу життя. Для бізнесу це також вигідно, адже можна залучати більше клієнтів і працювати не тільки на

місцевому, а й на міжнародному ринку [2]. Таким чином, цифровізація розширює можливості для розвитку підприємництва.

Ще одним важливим наслідком цифровізації є зміни на ринку праці. Зараз з'являється все більше нових форм зайнятості, таких як дистанційна робота або фриланс. Люди можуть працювати з будь–якого місця, маючи лише доступ до інтернету. Для студентів це особливо актуально, адже можна поєднувати навчання і роботу. Крім того, цифрові технології дозволяють створити власний бізнес з мінімальними витратами. Наприклад, можна відкрити інтернет–магазин або надавати послуги онлайн. Це значно спрощує старт підприємницької діяльності.

Окремо варто сказати про те, як цифровізація змінює повсякденне життя людей. Зараз більшість послуг доступні онлайн. Наприклад, можна оплатити комунальні послуги, замовити доставку їжі, купити квитки або навіть оформити документи через інтернет. Це значно економить час і робить життя більш комфортним. Також цифрові технології спрощують спілкування між людьми, адже можна швидко обмінюватися інформацією через соціальні мережі або месенджери.

Проте цифровізація має і негативні сторони. Однією з головних проблем є кібербезпека. З розвитком технологій зростає і кількість кібератак. Шахраї можуть отримати доступ до особистих даних або банківських рахунків. Тому дуже важливо дотримуватися правил безпеки в інтернеті та бути обережними при використанні онлайн–сервісів [1].

Ще однією проблемою є цифрова нерівність. Не всі люди мають однаковий доступ до інтернету та сучасних технологій. Наприклад, у деяких регіонах інтернет може бути повільним або взагалі відсутнім. Також не всі можуть дозволити собі дорогі пристрої. Через це частина населення має менше можливостей для навчання, роботи та розвитку.

Крім того, цифровізація впливає на зайнятість населення. З одного боку, з'являються нові професії, пов'язані з технологіями. Але з іншого боку, деякі традиційні професії поступово зникають, оскільки їх замінюють автоматизовані системи. Це означає, що людям потрібно постійно вдосконалювати свої знання та навички, щоб залишатися конкурентоспроможними на ринку праці.

З власного досвіду можу сказати, що цифрові технології значно допомагають у навчанні. Наприклад, використання Excel дозволяє швидко виконувати економічні розрахунки і краще розуміти матеріал. Також онлайн–платформи дають можливість отримувати доступ до великої кількості навчальних ресурсів. Це робить навчання більш зручним і ефективним.

Також хочу додати, що цифровізація впливає не лише на економіку, а й на мислення людей. Люди звикають до швидкого отримання інформації, стають більш мобільними та гнучкими. Це допомагає швидше адаптуватися до змін, але водночас вимагає більшої відповідальності при роботі з інформацією.

Отже, цифровізація економіки має дуже великий вплив на сучасне суспільство. Вона відкриває нові можливості для розвитку, спрощує життя людей і робить економіку більш ефективною. Проте разом із цим з'являються і нові виклики, такі як кібербезпека, нерівний доступ до технологій та зміни на ринку праці. Тому важливо навчитися правильно користуватися цифровими інструментами і постійно розвиватися. Лише за таких умов можна максимально використати переваги цифровізації та забезпечити подальший розвиток суспільства [3].

Список використаних джерел

1. Дашко І. М., Михайліченко Л. В. Цифровізація економіки як нова реальність України в умовах сьогодення. Економічний простір. 2024. № 190. С. 237–241. URL: https://prostir.pdaba.dp.ua/index.php/journal/article/view/1492 (дата звернення: 05.05.2026)

2. Кириленко С. В. Цифровізація економіки як рушій розвитку інтелектуального бізнесу та електронної комерції // Вісник Східноукраїнського національного університету імені Володимира Даля. 2025. № 3 (289). С. 21–29. DOI: 10.33216/1998–7927–2025–289–3–21–29. URL: https://journals.snu.edu.ua/index.php/VisnikSNU/article/view/1053 (дата звернення: 05.05.2026).

3. Филипенко О. М., Синицина Г. А. Основні тренди та перспективи цифровізації економіки України // Бізнес Інформ. 2023. № 3. С. 43–50. DOI: 10.32983/2222–4459–2023–3–43–50. URL: https://www.business–inform.net/article/?year=2023&abstract=2023_3_0_43_50 (дата звернення: 05.05.2026).

Корольова А.А.,
студентка 3 курсу ОПС «Фаховий молодший бакалавр»
спеціальності 121 «Інженерія програмного забезпечення»
освітньо-професійної програми «Інженерія програмного забезпечення»;
науковий керівник:
Савченко С. С.,
старший викладач,
циклова комісія з програмування
Фаховий коледж бізнесу та аналітики НАСОА

БІТОВИЙ ВЕКТОР ЯК КОМПАКТНА СТРУКТУРА ДАНИХ: ПОБІТОВІ ОПЕРАЦІЇ ТА ЗАСТОСУВАННЯ Д ЛЯ ПРЕДСТАВЛЕННЯ МНОЖИН

Решето Ератосфена є одним з найдавніших алгоритмів пошуку простих чисел. Для знаходження всіх простих до мільярда алгоритм потребує масив позначок розміром один мільярд елементів, де кожна позначка вказує, чи є відповідне число складеним. Якщо зберігати кожну позначку як окремий байт типу boolean, масив займе один гігабайт оперативної пам'яті, що є неприйнятним для більшості систем. Якщо ж пакувати по вісім позначок в один байт або по 64 в одне 64-бітне ціле число, розмір скорочується до 125 мегабайт. Саме ця ідея пакування булевих значень на рівні окремих бітів лежить в основі структури даних, що називається бітовим вектором [1].

Бітовий вектор є структурою даних, що зберігає послідовність булевих значень з максимальною щільністю упаковки: один біт на одне значення. Реалізація через масив 64-бітних цілих чисел (long) пакує 64 логічні елементи в одне слово. Для вектора з n елементів потрібно ceil(n/64) слів замість n байтів у boolean–масиві, що забезпечує 64-кратну економію пам'яті. Індекс потрібного слова обчислюється як i / 64, позиція біта всередині слова як i % 64. Суміжне зберігання слів у пам'яті і обробка одразу 64 елементів за одну машинну операцію роблять бітовий вектор ефективним не лише за пам'яттю, а й за швидкодією [2].

Доступ до окремого біта реалізується через операції бітової маски. Для встановлення біта з індексом i у значення 1 застосовується операція АБО зі словом, де встановлений лише біт на позиції i % 64: words[i/64] |= (1L << (i%64)). Очищення біта виконується через І з інвертованою маскою: words[i/64] &= ~(1L << (i%64)). Читання біта використовує І з маскою і перевірку результату на ненульовість: (words[i/64] >> (i%64)) & 1. Інверсія (toggle) виконується через XOR з маскою: words[i/64] ^= (1L << (i%64)). Усі чотири операції виконуються за сталий час O(1) і не потребують циклів або умовних переходів [3].

Ключова перевага бітового вектора проявляється при виконанні операцій не над окремими елементами, а над цілими словами одразу. Операція І двох бітових векторів виконується через побітове І відповідних слів масивів: один цикл з ceil(n/64) ітерацій обробляє весь вектор, на кожній ітерації паралельно опрацьовуючи 64 пари елементів за одну операцію. Аналогічно реалізуються АБО, XOR та інверсія. Для векторів розміром n загальна кількість машинних операцій дорівнює ceil(n/64) замість n при поелементній обробці. Сучасні процесори з підтримкою SIMD-розширень можуть обробляти ще більші блоки за одну операцію, що додатково підсилює перевагу бітового вектора у задачах масової булевої обробки.

Бітовий вектор є природною реалізацією математичної скінченної множини над універсальною множиною фіксованого розміру. Якщо бітовий вектор відповідає підмножині елементів з {0, 1, ..., n–1}, то встановлений біт на позиції і означає належність елемента і до множини. Тоді об'єднання двох множин відповідає побітовому АБО двох векторів, перетин відповідає побітовому І, різниця реалізується через І з інвертованим другим вектором, а симетрична різниця через XOR. Перевірка порожності множини зводиться до перевірки, чи всі слова вектора є нульовими. Підрахунок кількості елементів множини виконується через операцію popcount підрахунку встановлених бітів у слові, яку сучасні процесори підтримують на апаратному рівні. Усі операції над множинами виконуються за O(n/64).

Фільтр Блума є імовірнісною структурою даних, що базується на бітовому векторі і дозволяє перевіряти належність елемента до множини з гарантованою відсутністю хибно–негативних і контрольованим рівнем хибно-позитивних відповідей. При додаванні елемента k незалежних хеш-функцій обчислюють k індексів, і відповідні біти вектора встановлюються в 1. При перевірці ті ж хеш–функції обчислюють k індексів, і якщо хоча б один біт виявляється нульовим, елемент гарантовано відсутній. Фільтри Блума можуть використовуватися у веб–браузерах для перевірки шкідливих URL-адрес, у базах даних для уникнення непотрібних доступів до диска, а також у розподілених системах для ефективної синхронізації даних.

Бітовий вектор показує, що ефективність структури даних пов'язана не лише з асимптотичною складністю, але й з іншими постійними факторами, пов'язаними з апаратною реалізацією. Упаковка 64 логічних значень в одне машинне слово перетворює побітові операції на масивні паралельні обчислення без явного паралелізму в коді. Застосування бітового вектора у вигляді представлення множин, фільтра Блума і решета Ератосфена охоплює задачі від теоретико–числових обчислень до промислових систем фільтрації. Це робить бітовий вектор прикладом тісного зв'язку між математичними абстракціями і можливостями апаратного забезпечення.

Список використаних джерел

1. Кузьменко І. М., Дацюк О. А. Базові алгоритми та структури даних : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2022. 137 с.
2. Нестерко А. Б., Настенко Д. В., Труніна Г. О. Обчислювальна техніка та програмування. Лабораторні роботи. Ч. 1. Київ : КПІ ім. Ігоря Сікорського, 2020. 83 с.
3. Тарнавський Ю. А. Java–програмування : комп'ютерний практикум. Київ : КПІ ім. Ігоря Сікорського, 2021. 95 с.

Мельник О. В.,
студентка 2 курсу ОПС «Фаховий молодший бакалавр»
спеціальності 051 «Економіка»
освітньо–професійної програми «Бізнес статистика та аналітика»;
науковий керівник:
Пантєєв Р. Л.,
кандидат технічних наук,
доцент кафедри інформаційних технологій,
Національна академія статистики, обліку та аудиту

АЛГОРИТМИ СОРТУВАННЯ: КЛАСИФІКАЦІЯ, АНАЛІЗ СКЛАДНОСТІ ТА ПОРІВНЯЛЬНА ХАРАКТЕРИСТИКА

Задача впорядкування послідовності є однією з найчастіших у програмуванні: відсортовані дані є передумовою бінарного пошуку, спрощують злиття наборів, пришвидшують виявлення дублікатів і полегшують виведення. Незважаючи на те, що задача формулюється просто, існує десяток принципово різних підходів до її вирішення, і вибір між ними суттєво впливає на продуктивність залежно від обсягу даних і їх початкового стану [1].

Прості алгоритми – бульбашкове, сортування вибором і вставками – мають часову складність $O(n^2)$ у найгіршому випадку. Бульбашкове виконує n^2 порівнянь у будь-якому випадку. Сортування вибором завжди робить n^2 порівнянь, але лише $O(n)$ обмінів. Сортування вставками оптимальне для майже відсортованих масивів: на вже впорядкованих даних воно виконується за $O(n)$. Ці алгоритми прості у реалізації і ефективні для малих n [2].

Сортування злиттям (merge sort) використовує стратегію "розділяй і владарюй": масив ділиться навпіл, кожна половина сортується рекурсивно, потім дві відсортовані половини зливаються за $O(n)$. Загальна складність $O(n \log n)$ гарантована у всіх випадках. Алгоритм стабільний (рівні елементи

зберігають відносний порядок) і природно підходить для зовнішнього сортування великих файлів. Недолік – додаткова пам'ять O(n) для масиву злиття.

Швидке сортування (quicksort) обирає опорний елемент (pivot) і розподіляє всі менші елементи ліворуч, більші праворуч. Потім рекурсивно сортує обидві частини. Середня складність O(n log n), але у найгіршому випадку (завжди обираємо мінімум або максимум) – $O(n^2)$. Рандомізований вибір pivot або вибір медіани зі трьох зводить ймовірність деградації до мінімуму. Quicksort сортує на місці і має кращу кеш–поведінку, ніж merge sort [3].

Пірамідальне сортування (heapsort) будує з масиву двійкову купу за O(n), потім n разів вилучає максимальний елемент за O(log n). Загальна складність O(n log n) гарантована у всіх випадках, а додаткова пам'ять – O(1). На відміну від quicksort, heapsort не є рекурсивним і не страждає від стека викликів. Проте на практиці він повільніший за quicksort через погану кеш–локальність операцій купи.

Сортування підрахунком і порозрядне (counting і radix sort) не є алгоритмами порівняння і можуть подолати теоретичну нижню межу O(n log n). Сортування підрахунком за O(n + k), де k – діапазон значень, підраховує кількість входжень кожного значення і відновлює послідовність без порівнянь. Порозрядне сортування застосовує counting sort порозрядно і досягає O(n * d), де d – кількість цифр.

Стабільність сортування означає, що рівні елементи зберігають початковий відносний порядок. Merge sort і insertion sort є стабільними; quicksort і heapsort – нестабільні у стандартній реалізації. Стабільність важлива при сортуванні за кількома ключами: наприклад, спершу за прізвищем, потім за ім'ям. Сортування "на місці" (in-place) не потребує додаткової пам'яті пропорційно n – quicksort і heapsort задовольняють цю вимогу.

У Java метод Arrays.sort для примітивних масивів використовує dual-pivot quicksort, розроблений Владімиром Ярославцевим, який у середньому виконує на 20% менше порівнянь, ніж класичний quicksort. Для масивів об'єктів і колекцій застосовується TimSort – гібрид merge sort і insertion sort, оптимізований для реальних даних, що містять впорядковані підпослідовності. TimSort є стабільним і гарантує O(n log n).

Radix sort – алгоритм без порівнянь, що сортує цілі числа поцифрово від молодшого розряду до старшого. Він виконується за $O(d \cdot n)$, де d – кількість цифр у числі. При $d = O(\log n)$ це еквівалентно O(n log n), але з меншою константою порівняно з quicksort. Radix sort застосовується для сортування великих масивів цілих ідентифікаторів, IP-адрес і хеш-значень.

Зовнішнє сортування застосовується тоді, коли обсяг даних перевищує доступну оперативну пам'ять. Класичний підхід – зовнішнє злиття: дані читаються блоками, кожен блок сортується у пам'яті, відсортовані фрагменти

записуються на диск, потім зливаються за схемою k–way merge. Кількість проходів по диску є O(log k), де k – кількість початкових фрагментів.

Жоден окремий алгоритм сортування не є оптимальним для всіх задач: insertion sort перемагає на малих і майже впорядкованих масивах, merge sort є надійним для зовнішнього сортування і посилань, quicksort лідирує на випадкових даних у внутрішній пам'яті, а counting sort знімає обмеження O(n log n) для цілих чисел у вузькому діапазоні. Розуміння цих компромісів є необхідним для вибору правильного інструменту у конкретній задачі.

Список використаних джерел

1. Кузьменко І. М., Дацюк О. А. Базові алгоритми та структури даних : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2022. 137 с.
2. Грудзинський Ю. Є. Алгоритми та структури даних : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2022. 215 с.
3. Бугаєва Л. М., Ковалюк Д. О. Алгоритми та структури даних : комп'ютерний практикум. Київ : КПІ ім. Ігоря Сікорського, 2022. 34 с.

Нємов О. С.,
студент 3 курсу ОПС «Фаховий молодший бакалавр»
спеціальності 121 «Інженерія програмного забезпечення»
освітньо-професійної програми «Інженерія програмного забезпечення»;
науковий керівник:
Байдінов Д. О.,
старший викладач,
голова циклової комісії з програмування
Фаховий коледж бізнесу та аналітики НАСОА

ЧЕРВОНО–ЧОРНЕ ДЕРЕВО: САМОБАЛАНСУВАННЯ ТА ГАРАНТОВАНА СКЛАДНІСТЬ ОПЕРАЦІЙ

Бінарне дерево пошуку деградує до зв'язаного списку при вставці елементів у відсортованому порядку. Це перетворює O(log n) операції на O(n) і робить звичайний BST непридатним для виробничих систем з непередбачуваним порядком вхідних даних. Самобалансуючі дерева вирішують цю проблему через автоматичне перебудування структури після кожної модифікації. Червоно-чорне дерево є найпоширенішою реалізацією цієї ідеї [1].

Червоно-чорне дерево є бінарним деревом пошуку, де кожен вузол забарвлений у червоний або чорний колір. П'ять властивостей визначають коректне червоно-чорне дерево: кожен вузол є або червоним, або чорним; корінь завжди чорний; листові nil-вузли вважаються чорними; червоний вузол не може мати червоного нащадка; будь-який шлях від вузла до листового nil містить однакову кількість чорних вузлів [2].

Ці властивості спільно гарантують, що найдовший шлях від кореня до листа не більш ніж вдвічі довший за найкоротший. Оскільки найкоротший шлях складається лише з чорних вузлів, а найдовший чергує червоні і чорні, висота дерева з n вузлів не перевищує $2*\log_2(n+1)$. Це гарантує O(log n) для пошуку, вставки і видалення у будь-якому випадку.

Основним інструментом перебалансування є ротації – локальні перетворення, що змінюють структуру дерева без порушення інваріанти BST. Ліва ротація навколо вузла x робить правого нащадка у новим коренем, x стає лівим нащадком y, а колишній лівий нащадок y переходить до правого місця x. Права ротація є дзеркальною операцією. Ротація виконується за O(1).

При вставці новий вузол завжди забарвлюється в червоний колір. І якщо батьківський вузол чорний, властивості дерева не порушуються. Якщо батьківський вузол червоний, порушується четверта властивість, і алгоритм виправляє це через серію перефарбувань і обертань. Існує три випадки в залежності від кольору дядька вузла. У найгіршому випадку виправлення піднімається до кореня, але загальна кількість обертань не перевищує двох.

Видалення є складнішим, оскільки видалення чорного вузла порушує п'яту властивість. Алгоритм вводить "подвійний чорний" вузол і вирішує цю проблему через серію випадків, які або вирішують проблему локально, або зводять її до тієї, що знаходиться вище. І гарантія: не більше трьох обертань на видалення [3].

Червоний–чорний дерево менш збалансоване, ніж AVL-дерево: AVL має різницю у висоті менше одного між піддеревами, а червоно–чорне дерево дозволяє вдвічі більший розкид. Це означає трохи довші пошуки в червоно-чорному дереві, але значно менше ротацій при вставці і видаленні. Для задач з частими модифікаціями і рідшим пошуком червоно-чорне дерево виявляється ефективнішим.

У Java класи TreeMap і TreeSet реалізовані саме на основі червоно-чорного дерева. Кожен вузол внутрішнього класу Entry містить поле color типу boolean. Методи fixAfterInsertion і fixAfterDeletion виконують перебалансування після кожної модифікації. Такий вибір дає гарантовані O(log n) для всіх операцій на відміну від HashMap, де середнє O(1) може деградувати.

TreeMap не є потокобезпечним: одночасна модифікація з кількох потоків призводить до некоректного стану. Collections.synchronizedSortedMap загортає його у синхронізований wrapper, але блокує всі операції на один м'ютекс, усуваючи паралелізм читання. Для багатопотокових сценаріїв Java пропонує ConcurrentSkipListMap – структуру, що реалізує NavigableMap на основі списку з пропусками і забезпечує lock–free доступ при конкурентних читаннях і записах.

Червоно–чорні дерева широко використовуються поза межами Java. Планувальник CFS ядра Linux зберігає черги процесів у червоно-чорному дереві, де ключем є накопичений віртуальний час виконання. У C++ стандартна бібліотека реалізує std::map і std::set також на основі червоно-чорного дерева. Це свідчить про те, що структура залишається практичним вибором для задач, де потрібні гарантовані логарифмічні межі при інтенсивних операціях вставки і видалення.

B-дерева є узагальненням червоно-чорних дерев для дискових систем: вузол може мати до t нащадків, де t – порядок дерева. Це зменшує висоту дерева і кількість звернень до диска при пошуку у великих індексах. Більшість реляційних СУБД, зокрема PostgreSQL і MySQL InnoDB, використовують B+ дерева як структуру індексу: всі значення зберігаються у листях, а внутрішні вузли містять лише ключі для навігації.

Операція пошуку найближчого елемента (floor і ceiling) є природною для збалансованих BST. floor(k) повертає найбільший ключ, що не перевищує k; ceiling(k) – найменший ключ, що не менший за k. Ці операції реалізуються обходом дерева з відстеженням "кандидата" на кожному рівні і виконуються за O(log n). В Java вони доступні через NavigableMap.floorKey і ceilingKey.

Червоно-чорне дерево є прикладом структури даних, де складний інваріант підтримується у фоні для забезпечення гарантованої продуктивності. П'ять властивостей і алгоритми їх відновлення після модифікацій є результатом кількох десятиліть теоретичних досліджень. Розуміння механізму балансування червоно–чорного дерева є важливою основою для роботи з будь–якими самобалансуючими деревоподібними структурами даних.

Список використаних джерел

1. Марченко О. І., Марченко О. О. Структури даних та алгоритми : підручник. Ч. 1. Київ : Просвіта, 2024. 268 с.
2. Скулиш М. А., Суліма С. В. Алгоритми та структури даних : лабораторний практикум. Київ : КПІ ім. Ігоря Сікорського, 2021. 109 с.
3. Шилдт Г. Java. Повне керівництво. 10–те вид. Київ : Діалектика, 2019. 1376 с.

Павлюк Є. П.,
студент 3 курсу першого (бакалаврського) рівня вищої освіти
спеціальності 122 «Комп'ютерні науки»
освітньо–професійної програми «Комп'ютерні науки
та інформаційні технології»;
науковий керівник:
Пантєєв Р. Л.,
кандидат технічних наук,
доцент кафедри інформаційних технологій,
Національна академія статистики, обліку та аудиту

REST API ЯК АРХІТЕКТУРНИЙ СТИЛЬ: ПРИНЦИПИ ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ

До появи REST веб–сервіси будувались переважно на SOAP – протоколі з жорсткою схемою XML–повідомлень і складною специфікацією. У 2000 році Рой Філдінг у своїй докторській дисертації описав архітектурний стиль, що вже де-факто використовувала мережа Інтернет, і назвав його Representational State Transfer. REST не є протоколом чи стандартом – це набір архітектурних обмежень, що визначають, яким має бути дизайн розподіленої системи.

Філдінг сформулював шість обмежень REST. Клієнт-сервер: розмежування інтерфейсу користувача і зберігання даних. Відсутність стану: кожен запит містить всю інформацію для обробки, сервер не зберігає стан сесії між запитами. Кешованість: відповіді позначаються як кешовані або ні. Єдиний

інтерфейс: ресурси ідентифікуються URI, а взаємодія здійснюється через стандартні методи. Багаторівнева система і код на вимогу – необов'язкове шосте обмеження [1].

Центральним поняттям REST є ресурс – будь-яка іменована сутність, до якої є доступ. Ресурси ідентифікуються URI: /users, /users/42, /users/42/orders. URI має бути іменником (що?), а не дієсловом (що зробити?): /users правильно, /getUsers – ні. Ієрархія URI відображає відношення між ресурсами. Один ресурс може мати кілька представлень (JSON, XML, HTML) – клієнт вказує бажаний формат через заголовок Accept [2].

HTTP-методи визначають операцію над ресурсом. GET отримує ресурс: безпечний (не змінює стан) і ідемпотентний. POST створює новий ресурс або ініціює дію: не ідемпотентний. PUT замінює ресурс повністю: ідемпотентний. PATCH частково оновлює ресурс. DELETE видаляє ресурс: ідемпотентний. Ідемпотентність означає, що повторні однакові запити дають той самий результат, що й перший.

HTTP-коди стану повідомляють клієнта про результат запиту. 2xx означає успіх: 200 OK, 201 Created, 204 No Content. 3xx – перенаправлення. 4xx – помилка клієнта: 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 409 Conflict, 422 Unprocessable Entity. 5xx – помилка сервера: 500 Internal Server Error, 503 Service Unavailable. Коректне використання кодів стану – частина контракту API, якого очікують клієнти [3].

JSON є стандартом де-факто для тіла запитів і відповідей REST API. Заголовок Content–Type: application/json повідомляє сервер про формат тіла запиту, а Accept: application/json – про бажаний формат відповіді. Версіонування API вирішується через URI (/api/v1/users), заголовок (Accept-Version) або параметр запиту. Пагінація великих колекцій реалізується через параметри page і size або cursor–based підхід .

REST порівнюють з альтернативами. SOAP є строго типізованим, підтримує WS-Security і підходить для корпоративних систем із жорсткими вимогами. GraphQL дозволяє клієнту точно вказати, які поля він хоче отримати, що зменшує over-fetching і under-fetching. gRPC використовує Protocol Buffers і HTTP/2 для ефективного бінарного транспорту між сервісами. REST залишається найпоширенішим через простоту і підтримку браузерами.

У Java REST API реалізуються через Spring Boot або стандарт Jakarta EE JAX–RS. Анотація @RestController у Spring позначає клас, чиї методи повертають дані (а не ім'я view). @GetMapping, @PostMapping, @PutMapping, @DeleteMapping прив'язують методи до HTTP-методів і шляхів. @PathVariable витягує змінні з URI, @RequestBody десеріалізує тіло запиту в Java–об'єкт, @RequestParam – параметри рядка запиту.

Версіонування API є критичним для підтримки зворотної сумісності. Найпоширенішим підходом є версія в URI: /api/v1/users, /api/v2/users. Альтернатива – заголовок Accept: application/vnd.myapp.v2+json, що відповідає

принципу Content Negotiation HTTP. При зміні контракту відповіді без версіонування клієнти отримують несумісні дані, що спричиняє збої у виробничих середовищах.

Специфікація OpenAPI 3.0 (Swagger) дозволяє описувати контракт REST API у форматі YAML або JSON. З цього опису генерується інтерактивна документація, клієнтські SDK для різних мов і серверні заглушки. У Spring Boot інтеграція здійснюється через бібліотеку springdoc-openapi: анотації @Operation, @ApiResponse і @Schema доповнюють автоматично виявлені ендпоінти детальними описами і схемами.

REST API є домінуючим підходом до проектування інтерфейсів між компонентами розподілених систем. Його сила – у мінімалізмі: HTTP-методи, URI і коди стану утворюють простий але виразний словник для більшості задач взаємодії клієнта і сервера. Дотримання принципів REST забезпечує незалежне масштабування клієнта і сервера і полегшує документування та тестування API.

Список використаних джерел

1. Fielding R. T. Architectural Styles and the Design of Network–based Software Architectures : doctoral dissertation. University of California, Irvine, 2000. 162 p.

2. Татарчук Д. Д., Діденко Ю. В., Свєчніков Г. С. Об'єктно–орієнтоване програмування мовою Java : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2024. 153 с.

3. Шилдт Г. Java. Повне керівництво. 10–те вид. Київ : Діалектика, 2019. 1376 с.

Палій В.О.,
студент 3 курсу ОПС «Фаховий молодший бакалавр»
спеціальності 121 «Інженерія програмного забезпечення»
освітньо-професійної програми «Інженерія програмного забезпечення»;
науковий керівник:
Савченко С. С.,
старший викладач,
циклова комісія з програмування
Фаховий коледж бізнесу та аналітики НАСОА

АЛГОРИТМИ ПОШУКУ У ВЕЛИКИХ НАБОРАХ ДАНИХ: ПОРІВНЯЛЬНИЙ АНАЛІЗ БІНАРНОГО, ФІБОНАЧЧІ ТА ХЕШ-ПОШУКУ

Пошук елемента у наборі даних є однією з найпоширеніших операцій у програмуванні. Від ефективності пошуку залежить продуктивність пошукових систем, баз даних, компіляторів та будь-яких систем, що оперують великими колекціями. Жоден універсальний алгоритм пошуку не існує: вибір методу визначається структурою даних, частотою оновлень, вимогами до швидкодії та доступними ресурсами пам'яті. Дослідження порівняльних характеристик основних алгоритмів пошуку є практично необхідним для свідомого проектування програмних систем.

Лінійний пошук перевіряє елементи послідовно від першого до останнього і знаходить цільовий елемент або встановлює його відсутність за O(n) у найгіршому випадку. Метод не накладає жодних вимог на впорядкованість даних і є оптимальним для несортованих масивів невеликого розміру. Однак при n від десятків тисяч елементів лінійна залежність від розміру колекції робить метод неприйнятним для реальних систем. Саме ця обмеженість мотивує вивчення більш ефективних підходів до пошуку [1].

Бінарний пошук забезпечує O(log n) для відсортованого масиву. На кожному кроці порівнюється цільовий елемент з середнім елементом поточного діапазону: якщо вони рівні, пошук завершено; якщо цільовий менший, пошук продовжується в лівій половині; якщо більший, у правій. Кожна ітерація вдвічі скорочує простір пошуку, тому глибина рекурсії не перевищує log2(n). При n = 1 000 000 бінарний пошук потребує не більше 20 порівнянь проти мільйона при лінійному. Попередня вимога сортованості є єдиним обмеженням методу [2].

Пошук Фібоначчі є варіантом бінарного пошуку, що використовує числа Фібоначчі замість ділення навпіл для визначення точки розбиття діапазону. Числа Фібоначчі F(k) підбираються так, щоб F(k) було більшим або рівним розміру масиву n, і далі порівняння виконується з елементом на позиції F(k–2). Метод не використовує операцію ділення і покладається лише на додавання і

віднімання, що може давати перевагу на архітектурах з повільним діленням. Асимптотична складність залишається O(log n).

Хеш–пошук забезпечує O(1) у середньому випадку і є принципово відмінним підходом від порівняльних методів. Хеш-функція відображає ключ пошуку на індекс у масиві (хеш-таблиці), і при відсутності колізій доступ до елемента виконується за одне звернення до масиву. Колізії, коли різні ключі дають однаковий індекс, вирішуються методом ланцюжків або відкритою адресацією. У найгіршому випадку (усі елементи у одному ланцюжку) складність деградує до O(n), але при якісній хеш-функції та розумному коефіцієнті заповнення середній час залишається O(1) [3].

Порівняльний аналіз трьох методів виявляє чіткий розподіл за умовами застосування. Бінарний пошук є оптимальним для статичних відсортованих масивів, де вставки і видалення трапляються рідко. Пошук Фібоначчі застосовується у специфічних умовах, коли доступ до пам'яті є нерівномірним або поділ навпіл технічно ускладнений. Хеш-пошук є найшвидшим для динамічних колекцій з частими операціями вставки, видалення та пошуку, але потребує додаткової пам'яті і втрачає впорядкованість.

Клас SearchAlgorithms реалізує всі три методи у вигляді статичних параметризованих методів з уніфікованим інтерфейсом. Метод binarySearch приймає відсортований масив і цільовий елемент, повертаючи індекс або -1. Метод fibonacciSearch реалізує пошук Фібоначчі для масивів примітивних числових типів. Метод hashSearch використовує внутрішню HashMap<T, Integer> для зберігання відображення елементів на індекси і забезпечує O(1) пошук після O(n) попереднього заповнення.

Практичний вибір алгоритму пошуку визначається конкретними умовами задачі. Якщо дані рідко змінюються і потребують частого пошуку, масив сортується один раз і бінарний пошук використовується для всіх подальших запитів. Якщо дані динамічно оновлюються і пошук є критичним, хеш–таблиця забезпечує найкращу середню продуктивність. Коли важлива детермінована верхня межа часу відповіді, а не лише середнє значення, перевага надається бінарному пошуку з гарантованим O(log n) у найгіршому випадку. Для задач, де ключі розподілені рівномірно, інтерполяційний пошук може досягти O(log log n) у середньому, але деградує до O(n) у найгіршому.

Алгоритми пошуку є прикладом того, як вибір структури даних і методу кардинально змінює продуктивність системи. Знання характеристик кожного алгоритму, його передумов, середньої і гарантованої складності та поведінки при різних розподілах вхідних даних, є необхідною компетенцією інженера програмного забезпечення. Реалізація набору методів пошуку в єдиному класі дозволяє порівнювати їх на практичних даних, вимірювати реальні затримки на конкретних наборах і обирати найбільш відповідний метод для конкретного профілю навантаження.

Список використаних джерел

1. Кормен Т., Лейзерсон Ч., Рівест Р., Штайн К. Алгоритми: побудова і аналіз. Київ : Вільямс, 2019. 1328 с.
2. Седжвік Р., Уейн К. Алгоритми на Java. Київ : Діалектика, 2018. 848 с.
3. Марченко О. І., Марченко О. О. Структури даних та алгоритми : підручник. Ч. 1. Київ : Просвіта, 2024. 268 с.

Пархоменко М. І.,
студент 3 курсу ОПС «Фаховий молодший бакалавр»
спеціальності 121 «Інженерія програмного забезпечення»
освітньо-професійної програми «Інженерія програмного забезпечення»;
науковий керівник:
Савченко С. С.,
старший викладач,
циклова комісія з програмування
Фаховий коледж бізнесу та аналітики НАСОА

КЛАС ДЛЯ ОБРОБКИ ТЕКСТОВОЇ ІНФОРМАЦІЇ: РЕАЛІЗАЦІЯ РЯДКОВИХ ОПЕРАЦІЙ ЗАСОБАМИ JAVA

Обробка текстової інформації є однією з найпоширеніших задач у програмуванні: більшість програмних систем приймають, перетворюють і виводять дані у вигляді рядків. Парсинг конфігураційних файлів, валідація введення користувача, форматування звітів, аналіз журналів і побудова запитів до баз даних спільно потребують широкого набору рядкових операцій. Стандартна бібліотека Java є хорошим інструментарієм для роботи з рядками, але розробка власного класу CustomString дозволяє зрозуміти, як працює внутрішня реалізація цих механізмів.

Ключова особливість String у Java – незмінність: щойно об'єкт створено, жодна операція не здатна змінити його вміст. Те, що виглядає як зміна рядка, насправді щоразу породжує новий об'єкт. Така поведінка дає дві практичні переваги: по-перше, рядки безпечні для одночасного читання кількома потоками без жодної синхронізації; по-друге, JVM може зберігати однакові рядкові літерали в одній ділянці пам'яті, відомій як пул рядків, замість того щоб дублювати їх. Звідси важливе правило: оператор == для рядків порівнює посилання, а не символи, тому перевіряти вміст слід виключно через equals [1].

Для сценаріїв, де рядок змінюється багаторазово, незмінність String призводить до надмірного створення тимчасових об'єктів і навантаження на

збирач сміття. Клас StringBuilder надає змінний рядок у вигляді внутрішнього масиву символів з динамічним розширенням, аналогічним ArrayList. Операції append, insert і delete виконуються над внутрішнім масивом без створення нових об'єктів. StringBuffer є потокобезпечною версією StringBuilder з синхронізованими методами, проте за рахунок вищих накладних витрат [2].

Основні операції над рядками охоплюють широкий спектр перетворень. length повертає кількість символів, charAt звертається до символу за індексом за O(1). substring виділяє підрядок за початковим і кінцевим індексами. indexOf знаходить першу позицію підрядка наївним алгоритмом за O(n*m), де n і m є довжинами рядків. replace замінює всі входження підрядка, split розбиває рядок за роздільником, trim видаляє крайні пробіли. toLowerCase і toUpperCase перетворюють регістр символів відповідно до правил Unicode [3].

Клас CustomString розроблено для розширення функціональності стандартного String і кращого розуміння внутрішньої реалізації рядків. Внутрішнє представлення реалізовано через масив символів char[], що дозволяє виконувати операції безпосередньо над символами без проміжних перетворень. Методи valueOf, concat і toCharArray забезпечують взаємодію з стандартними Java-типами. Конструктор приймає рядок, масив символів або інший CustomString для ініціалізації.

Регулярні вирази є потужним інструментом для пошуку і трансформації рядків за шаблоном. У Java клас Pattern компілює регулярний вираз в автомат для ефективного застосування, а клас Matcher здійснює пошук за скомпільованим шаблоном. Метод matches перевіряє відповідність цілого рядка шаблону, find знаходить наступне входження, replaceAll замінює всі входження на рядок заміни. Використання регулярних виразів є стандартним підходом для валідації email-адрес, номерів телефонів та інших структурованих текстових форматів.

Продуктивність рядкових операцій суттєво залежить від правильного вибору класу. Конкатенація n рядків через оператор + у циклі створює O(n) тимчасових об'єктів і має загальну складність O(n^2). Використання StringBuilder.append у тому самому циклі зводить складність до O(n) і зменшує навантаження на збирач сміття. Компілятор Java автоматично замінює конкатенацію у простих виразах на StringBuilder, але в циклах ця оптимізація не застосовується, тому явне використання StringBuilder є рекомендованою практикою.

Клас String.format і методи форматування рядків є важливим інструментом для побудови шаблонних текстових повідомлень і звітів. Специфікатори формату %s, %d, %f, %n дозволяють вбудовувати значення різних типів у рядковий шаблон аналогічно printf у C. Клас java.util.Formatter, на якому базується String.format, підтримує локалізацію числових форматів і форматів дат через клас java.util.Locale. З Java 15 метод String.formatted() є альтернативою, що дозволяє ланцюгові виклики. Для побудови рядків з

колекцій клас StringJoiner і метод String.join забезпечують зручний спосіб конкатенації елементів через заданий роздільник без зайвих умовних перевірок.

Клас для обробки текстової інформації є прикладом застосування принципів об'єктно–орієнтованого програмування до фундаментальної задачі роботи з рядками. Реалізація власного CustomString поглиблює розуміння внутрішньої структури незмінних об'єктів, стратегій управління пам'яттю, алгоритмічної складності рядкових операцій і механізму інтернування рядків у JVM. Вибір між String, StringBuilder та Pattern визначається не зручністю API, а тим, яка внутрішня операція при цьому відбувається і яку ціну вона матиме при n викликах.

Список використаних джерел

1. Васильєв О. М. Програмування мовою Java. Київ : Богдан, 2019. 696 с.
2. Татарчук Д. Д., Діденко Ю. В., Свєчніков Г. С. Об'єктно–орієнтоване програмування мовою Java : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2024. 153 с.
3. Шилдт Г. Java. Повне керівництво. 10–те вид. Київ : Діалектика, 2019. 1376 с.

Пархут М. В.,
студент 3 курсу першого (бакалаврського) рівня вищої освіти
спеціальності 122 «Комп'ютерні науки»
освітньо-професійної програми «Комп'ютерні науки
та інформаційні технології»;
науковий керівник:
Пантєєв Р. Л.,
кандидат технічних наук,
доцент кафедри інформаційних технологій,
Національна академія статистики, обліку та аудиту

ІНДЕКСИ В РЕЛЯЦІЙНИХ БАЗАХ ДАНИХ: СТРУКТУРА, ТИПИ ТА ВПЛИВ НА ПРОДУКТИВНІСТЬ ЗАПИТІВ

Без індексів будь-який пошук у таблиці вимагає послідовного сканування кожного рядка, тобто повного сканування таблиці. Для таблиці з мільйоном записів це означало б мільйон порівнянь для кожного запиту. Індекс – це окрема структура даних, яка зберігає відсортовані значення одного або кількох стовпців таблиці з посиланнями на фактичні рядки, щоб база даних могла знаходити необхідні записи без сканування всієї таблиці.

Найпоширеніший тип індексу – це B-дерево. Це збалансоване дерево, де всі листові вузли мають однакову глибину. Пошук починається з кореня і з кожним кроком зменшує простір пошуку вдвічі – складність O(log n) на відміну від O(n) для повного сканування. Листові вузли в B-дереві зберігають значення ключа та вказівник на рядок таблиці, а також зв'язані між собою у зв'язному списку для ефективних запитів на діапазон [1].

Первинний ключ (PRIMARY KEY) автоматично отримує унікальний індекс. У PostgreSQL індекс є окремою структурою з посиланнями на файл купи таблиці. У MySQL InnoDB таблиця зберігається всередині B-дерева первинного ключа. Це називається кластеризованим індексом. Крім того, вторинні індекси створюються явно за допомогою команди CREATE INDEX і зберігають значення індексованого стовпця разом з посиланням на первинний ключ рядка [2].

Складений (composite) індекс охоплює кілька стовпців одразу. Порядок стовпців у ньому критичний: індекс на (прізвище, ім'я) допомагає запитам, що фільтрують за прізвищем або за прізвищем і ім'ям разом, але не допомагає, якщо фільтрується лише ім'я. Це правило лівого префікса: індекс корисний лише тоді, коли запит використовує стовпці, починаючи з першого.

Селективність індексу визначає, наскільки добре він фільтрує рядки. Індекс на стовпці з гендером (M/F) має низьку селективність – оптимізатор часто ігнорує його і виконує повне сканування, оскільки це швидше. Індекс на електронній пошті або UUID має високу селективність і забезпечує

максимальний приріст продуктивності. Загальне правило просте: чим більше унікальних значень у стовпці, тим кориснішим буде індекс.

Покриваючий індекс – це коли всі стовпці, необхідні для запиту, вже є в індексі. Тоді база даних не звертається до основної таблиці, а відповідає безпосередньо з індексу – менше дискових читань, швидше результат. Наприклад, індекс на (user_id, created_at, status) повністю покриває запит, який вибирає ці три поля з умовою на user_id [3].

EXPLAIN ANALYZE у PostgreSQL – це найпростіший спосіб зрозуміти, що база даних робить із запитом: чи використовує вона індекс, скільки рядків сканує, скільки часу це займає. Index Scan означає, що індекс спрацював; Seq Scan означає, що сканується вся таблиця. Bitmap Index Scan спочатку збирає покажчики на потрібні рядки через індекс і читає їх у фізичному порядку – уникаючи хаотичних стрибків по диску.

Крім B-дерева, існують інші типи індексів для специфічних завдань. Хеш-індекс оптимізований лише для перевірки рівності (=) і не підтримує діапазонні запити. Індекси GiST та SP–GiST використовуються для геометричних даних та повнотекстового пошуку. GIN (Generalized Inverted Index) корисний для масивів та полів JSONB: він зберігає окремі елементи контейнера і дозволяє швидко знаходити рядки, що містять конкретний елемент.

Кожен індекс має вартість: він займає місце на диску і уповільнює операції запису. З кожним INSERT, UPDATE або DELETE база даних оновлює всі індекси на відповідних стовпцях. Таблиця з десятьма індексами вставлятиме дані у п'ять разів повільніше, ніж таблиця без індексів. Тому індекси варто цілеспрямовано створювати на стовпцях, що фактично використовуються у WHERE, JOIN та ORDER BY найбільш часто виконуваних запитів.

У Java з Hibernate або Spring Data JPA індекси оголошуються через анотацію @Index у складі @Table. Наприклад, @Table(indexes = @Index(columnList = "email", unique = true)) створює унікальний індекс на стовпці email під час генерації схеми. Для складених індексів columnList містить кілька стовпців, розділених комами. У виробничому розгортанні індекси зазвичай генеруються скриптами міграції через Flyway або Liquibase, а не автоматичною генерацією схеми Hibernate.

Частковий індекс будується лише на підмножині рядків у таблиці. Наприклад, індекс на стовпці email з умовою WHERE is_active = true охоплює лише активних користувачів і займає набагато менше місця. Він корисний тоді, коли більшість запитів стосуються певного стану даних – індекс менший, оновлюється рідше і його легше зберігати в пам'яті.

Індекси є одним з найбільш використовуваних інструментів для оптимізації запитів у реляційних базах даних. Хороший індекс може перетворити запит, що виконується кілька секунд, на такий, що виконується за мілісекунди. Водночас надмірна кількість індексів деградує продуктивність запису і збільшує споживання пам'яті. Ефективна стратегія індексування

базується на аналізі реальних запитів через EXPLAIN ANALYZE і вимірюванні результату після кожної зміни.

Список використаних джерел

1. Чирков А. В., Павленко М. А. Системи управління базами даних : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2021. 184 с.
2. Мартін Р. К. Чистий код: створення, аналіз та рефакторинг. Київ : Діалектика, 2021. 464 с.
3. Шилдт Г. Java. Повне керівництво. 10–те вид. Київ : Діалектика, 2019. 1376 с.

Рогульчик Т. С.,
студент 3 курсу ОПС «Фаховий молодший бакалавр»
спеціальності 121 «Інженерія програмного забезпечення»
освітньо-професійної програми «Інженерія програмного забезпечення»;
науковий керівник:
Савченко С. С.,
старший викладач,
циклова комісія з програмування
Фаховий коледж бізнесу та аналітики НАСОА

AVL–ДЕРЕВО ЯК САМОБАЛАНСОВАНА СТРУКТУРА ДАНИХ: АЛГОРИТМИ БАЛАНСУВАННЯ ТА АНАЛІЗ СКЛАДНОСТІ ОПЕРАЦІЙ

Бінарне дерево пошуку (BST) забезпечує логарифмічну складність для операцій пошуку, вставки та видалення за умови збалансованості. Однак при послідовному додаванні впорядкованих елементів BST вироджується у лінійний список, і всі операції деградують до O(n). Наприклад, вставка послідовності 1, 2, 3, ..., n у звичайне BST утворює ланцюжок вузлів, де пошук будь-якого елемента потребує n кроків. Структура даних, що автоматично підтримує збалансованість після кожної модифікації, є необхідною передумовою для гарантованої ефективності операцій над деревом.

AVL-дерево є самобалансованим бінарним деревом пошуку, запропонованим Адельсон-Вельським та Ландісом у 1962 році. Для кожного вузла v фактор балансу bf(v) визначається як різниця висот правого і лівого піддерев і може приймати лише значення з множини {–1, 0, 1}. Якщо після вставки або видалення фактор балансу якогось вузла виходить за межі допустимого діапазону, виконується локальна перебудова дерева через операції

обертання для відновлення інваріанту AVL. Ця властивість гарантує, що висота дерева з n вузлів не перевищує 1,44 * log2(n) [1].

Для відновлення балансу використовуються чотири типи обертань. Мале праве обертання (LL) застосовується, коли вузол розбалансований через вставку в ліве піддерево лівого нащадка: лівий нащадок стає новим коренем підграфу. Мале ліве обертання (RR) є симетричним до LL. Ліво-праве обертання (LR) є комбінацією малого лівого і малого правого і застосовується при вставці в праве піддерево лівого нащадка. Право-ліве обертання (RL) є симетричним до LR. Кожне обертання виконується за O(1) і торкається лише фіксованої кількості вузлів [2].

Аналіз часової складності AVL-дерева показує, що всі три основні операції, пошук, вставка та видалення, гарантовано виконуються за O(log n) у найгіршому випадку. Пошук аналогічний пошуку у звичайному BST і потребує не більше висоти дерева кроків. Вставка виконує стандартний пошук місця за O(log n), додає вузол і потім повертається назад для перевірки та відновлення балансу: щонайбільше O(log n) перевірок і одне або два обертання. Видалення є найскладнішою операцією, але також гарантовано виконується за O(log n).

Клас CustomAVLTree<T> реалізує AVL–дерево з параметризованим типом елементів, де T розширює інтерфейс Comparable<T>. Внутрішній клас Node зберігає значення, посилання на лівий і правий нащадки та висоту вузла. Висота оновлюється після кожної операції як max(height(left), height(right)) + 1. Методи rotateLeft і rotateRight реалізують відповідні обертання, а метод rebalance визначає тип дисбалансу і викликає потрібну комбінацію обертань. Публічні методи insert, find і erase делегують роботу рекурсивним внутрішнім методам.

AVL-дерева знаходять широке застосування у системах, де вимагається гарантована швидкість пошуку в умовах активних вставок і видалень. Системи управління базами даних використовують збалансовані дерева як основу для індексів. Компілятори зберігають таблиці символів у деревах для швидкого пошуку ідентифікаторів. Файлові системи організовують каталоги у збалансованих деревах. Пошукові системи використовують дерева для побудови інвертованих індексів [3].

Порівняння AVL-дерева з іншими самобалансованими деревами виявляє його характерну особливість: строгий інваріант балансу забезпечує мінімально можливу висоту, що дає оптимальний час пошуку, але вимагає більшої кількості обертань при вставці і видаленні порівняно з червоно-чорним деревом. Червоно-чорне дерево допускає висоту до 2 * log2(n) і має меншу константу при операціях модифікації. На практиці AVL-дерево є кращим вибором для задач з переважанням пошуку над модифікацією.

Операція видалення у AVL-дереві є найскладнішою і вимагає кількох окремих кроків. Спочатку виконується стандартне видалення вузла BST: якщо вузол є листком, він просто вилучається; якщо він має одного нащадка,

нащадок займає його місце; якщо обох нащадків, вузол замінюється найменшим елементом правого піддерева (in-order successor), який потім видаляється зі свого початкового місця. Після видалення виконується підйом по дереву від місця видалення до кореня з перевіркою і відновленням балансу у кожному вузлі. На відміну від вставки, де достатньо одного обертання, видалення може потребувати O(log n) обертань уздовж шляху від видаленого вузла до кореня.

AVL-дерево є класичним прикладом структури даних, де додаткова метаінформація у кожному вузлі, а саме висота або фактор балансу, дозволяє підтримувати глобальний інваріант і гарантувати логарифмічну складність у найгіршому випадку. Концепція балансу та локального відновлення порядку через обертання є фундаментальною ідеєю, що лежить в основі всього сімейства самобалансованих дерев. Розуміння принципів роботи AVL-дерева є необхідною основою для вивчення складніших збалансованих структур, B-дерев та сучасних реалізацій у стандартних бібліотеках.

Список використаних джерел

1. Марченко О. І., Марченко О. О. Структури даних та алгоритми : підручник. Ч. 1. Київ : Просвіта, 2024. 268 с.
2. Кормен Т., Лейзерсон Ч., Рівест Р., Штайн К. Алгоритми: побудова і аналіз. Київ : Вільямс, 2019. 1328 с.
3. Татарчук Д. Д., Діденко Ю. В., Свєчніков Г. С. Об'єктно-орієнтоване програмування мовою Java : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2024. 153 с.

Рубель М. С.,
студент 3 курсу ОПС «Фаховий молодший бакалавр»
спеціальності 121 «Інженерія програмного забезпечення»
освітньо-професійної програми «Інженерія програмного забезпечення»;
науковий керівник:
Савченко С. С.,
старший викладач,
циклова комісія з програмування
Фаховий коледж бізнесу та аналітики НАСОА

ДЕК ЯК УЗАГАЛЬНЕННЯ СТЕКА І ЧЕРГИ: ОРГАНІЗАЦІЯ ДВОСТОРОННЬОЇ ЧЕРГИ ТА ЇЇ АЛГОРИТМІЧНЕ ЗАСТОСУВАННЯ

У практиці програмування трапляються задачі, де одна частина алгоритму потребує поведінки стека, а інша частина того самого алгоритму потребує поведінки черги. Наприклад, при обробці потоку даних необхідно додавати нові елементи в один кінець структури і водночас мати можливість видаляти з обох кінців залежно від умов задачі. Використання окремих стека і черги ускладнює синхронізацію між ними і дублює збереження даних. Стек і черга є, по суті, обмеженими версіями більш загальної структури, яка дозволяє додавати і вилучати елементи з будь–якого кінця. Саме такою структурою є дек, що включає стек і чергу як окремі часткові випадки і при цьому зберігає константну складність усіх операцій.

Дек (double-ended queue, двостороння черга) є лінійною структурою даних, що підтримує шість основних операцій. Push_front додає елемент на початок, push_back додає на кінець, pop_front вилучає і повертає перший елемент, pop_back вилучає і повертає останній, front і back дозволяють переглянути відповідні кінці без вилучення. Принципово важливо, що всі шість операцій виконуються за сталий час O(1), оскільки структура взаємодіє лише з крайніми елементами і ніколи не переглядає середину [1].

Реалізація деку через кільцевий буфер зберігає елементи у масиві фіксованої або динамічної ємності з двома індексами, head і tail. На відміну від кільцевого буфера черги, де head рухається лише вперед, а tail лише назад, у деку обидва індекси можуть рухатись у будь–якому напрямку. Push_front зменшує head за модулем ємності і записує елемент, push_back записує за tail і збільшує його; обидві операції виконуються без переміщення інших елементів. Коли буфер заповнений, виконується розширення масиву з копіюванням даних, амортизована вартість якого не порушує загальної O(1) складності для push. Суміжне розміщення елементів у пам'яті забезпечує кеш-локальність і робить кільцевий буфер оптимальним вибором для більшості практичних задач [2].

Альтернативою є реалізація через двозв'язний список, де кожен вузол містить покажчики на попередній і наступний елементи, а структура деку зберігає покажчики на голову і хвіст списку. Push_front і push_back додають вузол за відповідним кінцем за O(1), pop_front і pop_back вилучають і звільняють вузол також за O(1). Перевага двозв'язного списку полягає у відсутності потреби у попередньо виділеному масиві і природній підтримці довільного розміру без перерозподілу пам'яті. Недоліком є додаткова пам'ять на два покажчики кожного вузла і нижча кеш–ефективність через несуміжне розміщення даних [3].

Найвідомішим алгоритмічним застосуванням деку є задача максимуму у ковзному вікні. Задано масив з n елементів і вікно ширини k, що ковзає по масиву зліва направо; для кожного положення вікна потрібно знайти максимальний елемент. Наївний підхід переглядає k елементів для кожного положення і має часову складність O(n·k), що є неприйнятним для великих масивів. Рішення через монотонний дек підтримує у деку індекси елементів так, що відповідні значення утворюють спадну послідовність. При обробці нового елемента з хвоста деку видаляються всі індекси, чиї значення не перевищують новий елемент, адже вони вже ніколи не стануть максимумом. Індекси, які вийшли за межі поточного вікна, видаляються з початку. Оскільки кожен індекс додається і видаляється не більше одного разу, загальна складність алгоритму становить O(n).

Серед інших практичних застосувань двосторонньої черги, найцікавішим є кеш LRU (найменш нещодавно використаний). При доступі до елемента кешу він переміщується на початок двосторонньої черги як найщойно використаний, а якщо кеш переповнюється, найстаріший елемент видаляється з кінця. У поєднанні з хеш–таблицею для швидкого пошуку така схема забезпечує O(1) для всіх операцій кешу. Навігація браузера вперед і назад природно моделюється через дек, де pop_back повертає попередню сторінку, а push_front фіксує перехід на нову. Перевірка рядка на паліндром послідовно порівнює символи з голови і хвоста деку, рухаючись назустріч, поки кінці не зустрінуться або символи не збіжаться, що дає лінійну складність O(n) при одному проході по рядку.

Дек є прикладом структури даних, чия цінність визначається не унікальним принципом доступу, як у стека чи черги, а гнучкістю одночасного доступу до обох кінців. Ця гнучкість робить дек незамінним у задачах, що потребують ковзного вікна, двонапрямленої навігації або поєднання режимів LIFO і FIFO в одній структурі. Два класичних внутрішніх подання, кільцевий буфер і двозв'язний список, відповідають різним вимогам до пам'яті та продуктивності і разом охоплюють увесь спектр практичних сценаріїв.

Список використаних джерел

1. Скулиш М. А., Суліма С. В. Алгоритми і структури даних : лабораторний практикум. Київ : КПІ ім. Ігоря Сікорського, 2021. 109 с.
2. Грудзинський Ю. Є. Алгоритми та структури даних : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2022. 215 с.
3. Бугаєва Л. М., Ковалюк Д. О. Алгоритми та структури даних : комп'ютерний практикум. Київ : КПІ ім. Ігоря Сікорського, 2022. 34 с.

Садовнича А. І.,
здобувачка вищої освіти першого (бакалаврського) рівня,
спеціальність «Статистика цифрової економіки»;
науковий керівник:
Ставицький О. В.,
кандидат економічних наук, доцент,
завідувач кафедри інформаційних технологій,
Національна академія статистики, обліку та аудиту

ІТ ЯК ІНСТРУМЕНТ САМОВИРАЖЕННЯ ТА ЇХ ВПЛИВ НА ЕМОЦІЙНИЙ СТАН ЛЮДИНИ

Інформаційні технології у сучасному суспільстві стали невід'ємною частиною життя кожної людини [1]. Вони охоплюють практично всі сфери діяльності – освіту, науку, медицину, бізнес, творчість, професійну діяльність, комунікацію та дозвілля. Сьогодні важко уявити повсякденне життя без смартфонів, комп'ютерів, інтернету чи цифрових платформ. Інформаційні технології не лише полегшують виконання різноманітних завдань, а й відкривають широкі можливості для самовираження особистості. Завдяки ІТ люди можуть демонструвати власні здібності, ділитися ідеями, створювати творчі проєкти та спілкуватися з людьми з різних куточків світу. Водночас інформаційні технології суттєво впливають на емоційний стан людини, формуючи як позитивні, так і негативні психологічні наслідки [2].

Самовираження є важливою потребою людини, оскільки саме через нього особистість може проявити власну індивідуальність, творчий потенціал та світогляд. У минулому можливості для самовираження були значно обмеженішими, адже для того, щоб донести свої думки чи творчість до великої аудиторії, потрібні були значні ресурси та підтримка. Сьогодні ж завдяки інформаційним технологіям кожна людина може створювати власний контент та поширювати його через інтернет. Це можуть бути тексти, фотографії, відео, музика, графічний дизайн, цифрове мистецтво, комп'ютерні програми чи навіть власні онлайн-проєкти.

Одним із найбільш поширених способів самовираження є використання соціальних мереж. Такі платформи дозволяють людям ділитися власними думками, досягненнями, фотографіями та творчими роботами. Багато користувачів створюють блоги, тематичні сторінки або канали, де можуть розповідати про свої інтереси, хобі чи професійну діяльність. Для багатьох людей це стає способом знайти однодумців, отримати підтримку та реалізувати власний творчий потенціал. Наприклад, художники можуть публікувати свої роботи онлайн, музиканти – записувати та поширювати власні композиції, а письменники – публікувати тексти та отримувати відгуки від читачів.

Однак самовираження через інформаційні технології не обмежується лише соціальними мережами чи розважальним контентом. Важливу роль ІТ відіграють у сфері освіти та професійної діяльності. Сучасні студенти активно використовують цифрові платформи для створення презентацій, навчальних проєктів, дослідницьких робіт та онлайн-портфоліо. Завдяки цьому вони можуть демонструвати власні знання, навички та креативність. Викладачі також застосовують інформаційні технології для створення інтерактивних матеріалів, проведення онлайн-лекцій та комунікації зі студентами. Таким чином, навіть освітня діяльність стає формою самовираження.

У професійній сфері інформаційні технології відкривають широкі можливості для творчості та розвитку. Програмісти створюють вебсайти, мобільні додатки та комп’ютерні програми, дизайнери працюють над графічними проєктами, а спеціалісти з маркетингу використовують цифрові інструменти для створення рекламних кампаній. Кожен із цих напрямів дозволяє людині проявити власний стиль мислення, професійні здібності та творчий підхід до роботи. Багато професій сьогодні безпосередньо пов’язані з цифровими технологіями, тому ІТ стали важливим інструментом реалізації особистісного потенціалу.

Особливе значення інформаційні технології мають для розвитку комунікації між людьми. Сучасні онлайн-платформи дозволяють підтримувати зв’язок із друзями, родиною та колегами незалежно від відстані. Люди можуть швидко обмінюватися повідомленнями, проводити відеодзвінки, брати участь у конференціях та працювати над спільними проєктами. Це особливо важливо в умовах глобалізації, коли співпраця між людьми з різних країн стає звичним явищем. Інформаційні технології сприяють об’єднанню людей, обміну культурним досвідом та розвитку міжнародного співробітництва.

Разом із можливостями для самовираження інформаційні технології значно впливають на емоційний стан людини. З одного боку, цей вплив є позитивним. Використання цифрових технологій часто допомагає людям відчувати себе впевненішими та успішнішими. Коли людина створює певний цифровий продукт і бачить результати своєї роботи, це викликає почуття задоволення, гордості та мотивації до подальшого розвитку. Наприклад, студент, який успішно створив презентацію чи реалізував навчальний проєкт,

може відчути впевненість у власних силах та бажання вдосконалювати свої навички.

Інформаційні технології також допомагають людям знаходити нові можливості для розвитку та самореалізації. Онлайн-курси, вебінари та освітні платформи роблять знання доступнішими для широкого кола людей. Людина може самостійно обирати темп навчання, здобувати нові навички та професії. Це позитивно впливає на емоційний стан, оскільки сприяє відчуттю особистісного розвитку. Крім того, доступ до великої кількості інформації допомагає людям швидше вирішувати проблеми та знаходити відповіді на важливі питання.

Важливим позитивним аспектом є й те, що інформаційні технології можуть знижувати рівень стресу у певних ситуаціях. Наприклад, онлайн-сервіси дозволяють економити час на виконання різних завдань: оплату рахунків, покупки, навчання чи роботу. Це робить життя більш комфортним та організованим. Під час дистанційного навчання або роботи людина має можливість самостійно планувати свій графік, що також може позитивно впливати на емоційний стан.

Проте вплив інформаційних технологій не завжди є позитивним. Однією з головних проблем сучасного цифрового середовища є надмірне інформаційне навантаження. Щодня людина отримує величезну кількість повідомлень, новин та інформації, яку складно повністю обробити. Постійний потік даних може викликати втому, зниження концентрації уваги та емоційне виснаження. Особливо це стосується студентів та працівників, які змушені одночасно виконувати багато завдань і швидко адаптуватися до нових технологій.

Ще однією проблемою є залежність від цифрових пристроїв та соціальних мереж. Багато людей проводять значну частину свого часу в інтернеті, що може негативно впливати на психологічний стан. Постійна потреба перевіряти повідомлення, переглядати стрічку новин або отримувати схвалення у вигляді лайків формує емоційну залежність від онлайн-середовища. У разі відсутності уваги чи підтримки людина може відчувати розчарування, тривогу або невпевненість у собі.

Особливо небезпечним є явище порівняння себе з іншими у цифровому просторі. У соціальних мережах люди зазвичай демонструють лише найкращі моменти свого життя, успіхи та досягнення. Через це інші користувачі можуть починати вважати власне життя менш успішним чи цікавим. Таке порівняння часто призводить до зниження самооцінки, невпевненості та емоційного дискомфорту, особливо серед молоді.

Крім того, надмірне використання інформаційних технологій може негативно впливати на живе спілкування між людьми. Хоча цифрова комунікація дозволяє швидко підтримувати зв’язок, вона не завжди здатна повністю замінити особисте спілкування. Люди можуть ставати менш емоційно відкритими, рідше проводити час разом у реальному житті та втрачати навички

безпосередньої взаємодії. Це інколи призводить до відчуття самотності та емоційної ізоляції.

Важливим прикладом впливу ІТ є використання цифрових платформ у командній роботі. Сьогодні люди з різних країн можуть працювати над одним проєктом, спілкуватися онлайн та обмінюватися ідеями. Це сприяє розвитку відповідальності, комунікативних навичок та відчуття причетності до глобальних процесів. Робота в міжнародних командах дозволяє людям відчувати себе частиною великої спільноти, що позитивно впливає на мотивацію та емоційний стан.

Разом із тим дистанційна робота та навчання можуть мати і певні недоліки. Постійне перебування перед екраном комп'ютера часто викликає втому, порушення режиму дня та емоційне перевантаження. Люди можуть втрачати баланс між роботою та відпочинком, оскільки цифрові технології дозволяють залишатися на зв'язку практично постійно. Через це зростає ризик професійного вигорання та психологічного стресу.

Для того щоб інформаційні технології приносили більше користі, важливо використовувати їх усвідомлено та відповідально. Людина повинна вміти контролювати час, проведений у цифровому середовищі, робити перерви та приділяти увагу реальному спілкуванню. Важливим є також розвиток цифрової культури та критичного мислення, що допоможе уникати негативного впливу інформаційного простору.

Отже, інформаційні технології є важливим інструментом самовираження та розвитку особистості. Вони відкривають широкі можливості для творчості, освіти, професійної діяльності та комунікації. Завдяки ІТ люди можуть реалізовувати власні ідеї, створювати нові проєкти та взаємодіяти з іншими людьми у глобальному середовищі. Водночас інформаційні технології мають значний вплив на емоційний стан людини. Вони можуть підвищувати мотивацію, впевненість та задоволення від діяльності, але також здатні викликати втому, стрес та емоційне виснаження. Саме тому важливо підтримувати баланс між цифровим і реальним життям, використовуючи інформаційні технології як засіб розвитку, а не як джерело постійного психологічного навантаження [3].

Список використаних джерел

1. Міністерство цифрової трансформації України. Дія. Цифрова освіта : вебсайт. URL: https://osvita.diia.gov.ua/ (дата звернення: 06.05.2026).
2. UNICEF Ukraine. Соціальні мережі та психічне здоров'я підлітків : вебсайт. URL: https://www.unicef.org/ukraine/ (дата звернення: 06.05.2026).
3. Український інститут розвитку освіти. Цифрова грамотність та безпечне використання інформаційних технологій : вебсайт. URL: https://uied.org.ua/en/ (дата звернення: 06.05.2026).

Слободяник Б. ,
студент 3 курсу ОПС «Фаховий молодший бакалавр»
спеціальності 121 «Інженерія програмного забезпечення»
освітньо-професійної програми «Інженерія програмного забезпечення»;
науковий керівник:
Байдінов Д. О.,
старший викладач,
голова циклової комісії з програмування
Фаховий коледж бізнесу та аналітики НАСОА

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ: ПРИНЦИПИ ІНКАПСУЛЯЦІЇ, НАСЛІДУВАННЯ ТА ПОЛІМОРФІЗМУ

До впровадження об'єктно-орієнтованої парадигми програмісти структурували свій код навколо функцій і процедур, які працювали з загальними даними або даними, переданими як аргументи. Але з ростом розмірів програм цей підхід зазнав невдачі: логічно пов'язані дані та функції зберігалися в різних частинах системи; зміни в одному модулі непередбачувано впливали на інший. Об'єктно–орієнтоване програмування запропонувало інший спосіб мислення: замість функцій і даних – об'єкти, які їх поєднують, взаємодіючи через визначений інтерфейс.

Центральним поняттям ООП є клас – шаблон, що описує структуру і поведінку певної сутності. Клас визначає поля (атрибути стану) і методи (операції над цим станом). Об'єкт є конкретним екземпляром класу, що займає власну ділянку пам'яті зі своїми значеннями полів. Один клас може породжувати необмежену кількість об'єктів, кожен з яких поводиться за одними правилами, але зберігає власний стан [1].

Інкапсуляція є першим з фундаментальних принципів ООП: приховування внутрішнього стану об'єкта від зовнішнього коду. У Java поля класу оголошуються з модифікатором private, а доступ до них надається через публічні методи доступу. Це розділення дозволяє змінювати внутрішню реалізацію класу без впливу на код, який його використовує: поки публічний інтерфейс зберігається, клієнтський код буде працювати правильно. Наслідування дозволяє створювати ієрархію класів, де дочірній клас успадковує всі поля та методи батьківського класу і може доповнювати або перевизначати їх за потреби. В Java успадкування реалізується ключовим словом extends. Мова підтримує лише одиночне успадкування класів, що уникає неоднозначностей ромбоподібної проблеми, характерних для мов з множинним успадкуванням. Конструктор підкласу звертається до конструктора предка через super, що гарантує коректну ініціалізацію успадкованого стану.

Поліморфізм дає змогу обробляти об'єкти різних класів через єдиний інтерфейс. Метод, оголошений у базовому класі або інтерфейсі, може бути

перевизначений у кожному підкласі, і виклик цього методу через посилання базового типу автоматично спрямовується до реалізації фактичного класу об'єкта. Цей механізм динамічного зв'язування дозволяє розширювати систему новими типами без зміни існуючої логіки [2].

Абстракція дозволяє визначити контракт окремо від реалізації. В Java цю роль виконують абстрактні класи та інтерфейси. Абстрактний клас може містити і абстрактні методи без тіла, і конкретні реалізації спільної логіки. Інтерфейс описує контракт поведінки без стану; клас може реалізовувати кілька інтерфейсів одночасно, що дає практичний шлях до множинного поліморфізму без ускладнень множинного успадкування [3].

П'ять принципів SOLID систематизують правила проектування класів. Single Responsibility – клас має лише одну причину для зміни. Open/Closed – відкритий для розширення, закритий для модифікації. Liskov Substitution – підклас замінює базовий без порушення поведінки. Interface Segregation – краще кілька вузьких інтерфейсів, ніж один широкий. Інверсія залежностей – залежності будуються через абстракції, а не через конкретні класи.

У Java ООП реалізується як і раніше: весь виконуваний код знаходиться в методах класів, і кожен клас успадковує від Object, кореня ієрархії. Анотація @Override означає, що якщо метод предка не оголошений, розробник перевизначає його замість того, щоб оголошувати новий з неправильною сигнатурою. Стандартні бібліотеки побудовані навколо інтерфейсів Collection, Comparable та Iterable, щоб їх можна було розширювати без зміни клієнтського коду.

Композиція є альтернативою успадкуванню і часто є кращим вибором. Замість того, щоб підклас успадковував поведінку батька, він містить посилання на об'єкт потрібного типу і делегує виконання операцій йому. Це зменшує зв'язок між класами і дозволяє змінювати поведінку об'єкта під час виконання, замінюючи залежність, що є основним принципом шаблону проектування Стратегія.

ООП і тестування тісно пов'язані: добре спроектовані класи природно підходять для модульного тестування. Залежності, передані через конструктор або сеттер, можуть бути замінені тестовими заглушками (моками), тому клас можна тестувати ізольовано від зовнішніх систем. Інтерфейси відіграють тут ключову роль: якщо клас залежить від інтерфейсу, а не від конкретної реалізації, заміна залежності в тесті стає тривіальною.

ООП – це не просто синтаксичні конструкції, а спосіб мислення про програму як про систему взаємодіючих об'єктів з визначеними обов'язками. Три стовпи – інкапсуляція, успадкування і поліморфізм – у поєднанні з принципами SOLID дають архітектурну основу для побудови систем, які легко розширювати, тестувати і підтримувати. Саме тому об'єктно-орієнтована парадигма залишається домінуючою в промисловій розробці на Java і суміжних мовах.

Список використаних джерел

1. Татарчук Д. Д., Діденко Ю. В., Свєчніков Г. С. Об'єктно–орієнтоване програмування мовою Java : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2024. 153 с.
2. Васильєв О. М. Програмування мовою Java. Київ : Богдан, 2019. 696 с.
3. Шилдт Г. Java. Повне керівництво. 10–те вид. Київ : Діалектика, 2019. 1376 с.

Ставицький І. О.,
студент 2 курсу першого (бакалаврського) рівня вищої освіти
спеціальності F2 «Інженерія програмного забезпечення»
освітньо-професійної програми «Інженерія програмного забезпечення»;
Київського національного університету імені Тараса Шевченка;
науковий керівник:
Пантєєв Р. Л.,
кандидат технічних наук,
доцент кафедри інформаційних технологій,
Національна академія статистики, обліку та аудиту

БІНАРНЕ ДЕРЕВО ПОШУКУ: СТРУКТУРА, ОПЕРАЦІЇ ТА АНАЛІЗ СКЛАДНОСТІ

Масиви та зв'язані списки вирішують проблему зберігання послідовностей, але мають спільний недолік: пошук елемента за значенням вимагає лінійного перегляду. Дерева, як клас структур даних, пропонують інший спосіб організації – ієрархічний, де кожен вузол з'єднаний з кількома нащадками. Бінарне дерево пошуку є найпростішою деревоподібною структурою, яка підтримує ефективний пошук, вставку та видалення, оскільки елементи розміщені у правильному порядку [1].

Бінарне дерево пошуку (BST) – це бінарне дерево, в якому кожен вузол зберігає ключ і посилання на лівого та правого нащадків. Основна властивість: усі ключі в лівому піддереві вузла строго менші за ключ вузла, а всі ключі в правому піддереві строго більші. Ця властивість зберігається для кожного вузла в дереві, і порядок, у якому вони шукаються, відповідає дереву [2].

Пошук починається з кореня, і на кожному кроці порівнюється бажаний ключ з ключем поточного вузла. Якщо вони рівні, елемент знайдено. Якщо бажаний ключ менший, рекурсія продовжується в лівому піддереві; якщо

більший, у правому. Кожне порівняння виключає половину дерева з подальшого пошуку, тому складність пропорційна висоті дерева h: O(h).

Операція вставки також схожа на пошук: це місце, де новий елемент розміщується на порожньому місці, де він би знаходився під час пошуку. Алгоритм порівнює ключ нового елемента з поточним вузлом і рухається вліво або вправо, поки не досягне нульового вказівника – де розташований новий лист. Складність вставки, як і пошуку, становить O(h).

Видалення є найскладнішою з трьох операцій і розглядає три випадки. Якщо вузол є листом його просто видаляють. Якщо вузол має одного нащадка батьківський вузол перенаправляє посилання на цього нащадка. Якщо вузол має двох нащадків його заміняють найменшим вузлом правого піддерева (або найбільшим лівого), після чого видаляють цей замінник зі свого початкового місця [3].

Обхід дерева в порядку inorder (лівий нащадок, корінь, правий нащадок) відвідує всі вузли у зростаючому порядку ключів. Ця властивість є прямим наслідком інваріанти BST і робить inorder–обхід ефективним способом отримати відсортовану послідовність за O(n) без додаткового сортування. Preorder–обхід корисний для серіалізації дерева, postorder для звільнення пам'яті.

Складність операцій BST залежить від висоти h, яка коливається від O(log n) для збалансованого дерева до O(n) для виродженого. Вироджений випадок виникає тоді, коли елементи додаються у відсортованому порядку: кожен новий вузол стає правим нащадком попереднього, і дерево набуває форми лінійного ланцюжка. За своєю структурою таке дерево є еквівалентним зв'язаному списку, а час пошуку деградує до O(n) – і жодної переваги над списком не залишається.

В Java відсортований асоціативний масив на основі BST реалізується класом java.util.TreeMap, що реалізує інтерфейс NavigableMap. TreeMap гарантує O(log n) для put, get і remove. Методи firstKey і lastKey повертають мінімальний і максимальний ключі, floorKey і ceilingKey знаходять найближчий ключ знизу і зверху від заданого значення, що є унікальними можливостями, недоступними у HashMap.

Запити на діапазон є природним застосуванням BST: знайти всі елементи, чиї ключі потрапляють у відрізок [a, b]. Алгоритм спускається до вузла з ключем a і виконує inorder–обхід до досягнення b, що займає O(h + k), де k – кількість знайдених елементів. Методи subMap, headMap і tailMap класу TreeMap реалізують саме такі запити, повертаючи NavigableMap з елементами у заданому діапазоні.

Просторова складність BST становить O(n): кожен з n вузлів зберігає ключ, значення і два покажчики на нащадків. На відміну від масиву, BST не потребує суміжного блоку пам'яті, тому вставка ніколи не викликає перерозподілу. Разом з тим розкидані у пам'яті вузли знижують кеш–

ефективність порівняно з масивом: кожен перехід до нащадка – потенційний промах кеш–лінії.

Бінарне дерево пошуку є концептуальним фундаментом для розуміння більш складних деревоподібних структур – AVL–дерев, червоно–чорних дерев, B–дерев. Інваріанта BST і три базові операції залишаються однаковими для всіх них: відмінність лише у механізмах підтримки балансу, що забезпечують гарантовану логарифмічну складність незалежно від порядку вставки.

Список використаних джерел

1. Марченко О. І., Марченко О. О. Структури даних та алгоритми : підручник. Ч. 1. Київ : Просвіта, 2024. 268 с.
2. Кузьменко І. М., Дацюк О. А. Базові алгоритми та структури даних : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2022. 137 с.
3. Грудзинський Ю. Є. Алгоритми та структури даних : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2022. 215 с.

Телетьон М. В.,
студент 3 курсу ОПС «Фаховий молодший бакалавр»
спеціальності 121 «Інженерія програмного забезпечення»
освітньо-професійної програми «Інженерія програмного забезпечення»;
науковий керівник:
Байдінов Д. О.,
старший викладач,
голова циклової комісії з програмування
Фаховий коледж бізнесу та аналітики НАСОА

ХЕШУВАННЯ ТА ШИФРУВАННЯ: ТЕОРЕТИЧНІ ОСНОВИ ТА ЗАСТОСУВАННЯ В ПРОГРАМНИХ СИСТЕМАХ

Будь-яка інформаційна система, яка зберігає або передає дані, стикається з проблемою їх захисту. Паролі, платіжні дані, медичні записи – все це має бути перетворено в щось, до чого ніхто не може отримати доступ або може перевірити як дійсне без розкриття того, що це таке. Хешування та шифрування вирішують ці дві задачі і є фундаментальними криптографічними примітивами, на яких захищені сучасні програмні системи.

Хеш-функція відображає дані будь-якого розміру в рядок фіксованої довжини. Криптографічна хеш-функція має три ключових властивості: детермінованість (однакові вхідні дані дають однаковий хеш), однобічність (відновити вхід за хешем обчислювально неможливо) і лавинний ефект (зміна одного біта вхідних даних повністю змінює результуючий хеш). Ці властивості роблять хеш–функції основою перевірки цілісності даних [1].

MD5 і SHA-1 вважалися стандартними до виявлення практичних колізійних атак і нині не рекомендовані для криптографічних цілей. SHA-256 та SHA-3, стандартизовані NIST, є прийнятними для перевірки цілісності файлів та цифрових підписів. Щодо зберігання паролів, жоден з них не підходить: вони занадто швидкі, і зловмисник може спробувати мільярди варіантів за секунду. Bcrypt, scrypt та Argon2 спеціально розроблені для паролів. Вони штучно уповільнені та мають механізм солі.

Симетричне шифрування використовує один і той самий ключ для шифрування та дешифрування. AES (Advanced Encryption Standard), стандартизований NIST, є найпоширенішим симетричним шифром [2]. Він оперує блоками 128 біт і підтримує ключі довжиною 128, 192 або 256 біт. Режим роботи CBC вимагає вектора ініціалізації (IV) для унеможливлення однакового шифртексту для однакових блоків відкритого тексту. Режим GCM додає автентифікацію цілісності до шифрування.

Асиметричне шифрування використовує пару ключів: публічний і приватний. Повідомлення, зашифроване публічним ключем, можна

розшифрувати лише відповідним приватним, і навпаки. RSA є найвідомішим алгоритмом асиметричного шифрування: його стійкість базується на обчислювальній складності факторизації великих цілих чисел. Асиметричне шифрування значно повільніше за симетричне, тому на практиці його використовують для обміну симетричним ключем.

Гібридне шифрування поєднує переваги обох підходів. Протокол TLS, що захищає HTTPS–з'єднання, використовує асиметричне шифрування лише для рукостискання: клієнт і сервер погоджують спільний симетричний ключ сесії, і весь подальший трафік шифрується AES–ом. Такий підхід забезпечує і безпеку обміну ключем, і продуктивність масового шифрування даних.

Цифровий підпис застосовує асиметричну криптографію у зворотному напрямку: відправник підписує хеш документа своїм приватним ключем, а отримувач перевіряє підпис публічним. Це гарантує автентичність (підпис міг зробити лише власник приватного ключа) і цілісність (будь-яка зміна документа інвалідує підпис). Сертифікати X.509, на яких базується PKI, є підписаними документами, що пов'язують публічний ключ з ідентифікатором.

В Java криптографічні операції доступні через пакети java.security і javax.crypto. Клас MessageDigest реалізує хешування: MessageDigest.getInstance("SHA–256") повертає екземпляр для обчислення хешу. Клас Cipher реалізує шифрування: Cipher.getInstance("AES/GCM/NoPadding") налаштовує алгоритм, режим і підбивку. KeyPairGenerator генерує пари ключів для RSA. Клас Signature реалізує підписання і перевірку [3].

Веселкові таблиці (rainbow tables) є заздалегідь обчисленими хешами мільярдів поширених паролів, що дозволяють миттєво знайти пароль за його хешем. Сіль (salt) – випадковий рядок, що додається до пароля перед хешуванням, повністю нейтралізує цей метод: однаковий пароль із різною сіллю дає різний хеш, і зловмисник не може використати передобчислену таблицю. Функції bcrypt і Argon2 генерують сіль автоматично і зберігають її разом з хешем.

Управління ключами є одним із найскладніших аспектів криптографії на практиці. Безпека зашифрованих даних повністю залежить від секретності ключа: скомпрометований ключ робить шифрування марним. У промислових системах ключі зберігаються в спеціалізованих сховищах: Java KeyStore, HashiCorp Vault або хмарних сервісах ключів. Ротація ключів – регулярна заміна активних ключів новими – обмежує збитки у разі компрометації.

Хешування і шифрування вирішують принципово різні задачі: хеш перевіряє цілісність і зберігає паролі, шифрування приховує вміст і забезпечує конфіденційність. Застосування цих примітивів разом із правильними алгоритмами, достатніми розмірами ключів і коректною реалізацією є необхідною умовою захищеності будь-якої сучасної програмної системи.

Список використаних джерел

1. Шилдт Г. Java. Повне керівництво. 10–те вид. Київ : Діалектика, 2019. 1376 с.
2. National Institute of Standards and Technology. Federal Information Processing Standard 197 : Advanced Encryption Standard (AES). Gaithersburg : NIST, 2001. 51 p.
3. National Institute of Standards and Technology. Federal Information Processing Standard 180–4 : Secure Hash Standard (SHS). Gaithersburg : NIST, 2015. 35 p.

Ціцура П.В.,
студент 3 курсу ОПС «Фаховий молодший бакалавр»
спеціальності 121 «Інженерія програмного забезпечення»
освітньо-професійної програми «Інженерія програмного забезпечення»;
науковий керівник:
Савченко С. С.,
старший викладач,
циклова комісія з програмування
Фаховий коледж бізнесу та аналітики НАСОА

ВЕКТОР ЯК СТРАТЕГІЯ РОЗШИРЕННЯ ДИНАМІЧНОГО МАСИВУ ТА АНАЛІЗ СКЛАДНОСТІ ОПЕРАЦІЙ

Серед усіх базових структур даних довільний доступ до елемента за цілочисельним індексом є особливо цінним. Будь-який елемент послідовності може бути доступний за постійний час O(1) незалежно від її розміру, і існує безліч алгоритмів, які потребують швидкого читання та запису в будь-яку позицію. Статичний масив враховує це, зберігаючи елементи в пам'яті послідовно таким чином, що адреса елемента з індексом i задається як base + i * sizeof(element). Фіксований розмір статичного масиву є серйозним обмеженням, оскільки кількість елементів невідома в більшості задач. Усунення цього обмеження без втрати доступу O(1) є головним завданням динамічного масиву, або вектора.

Вектор є абстрактною структурою даних, яка моделює послідовність елементів змінного розміру і підтримує довільний доступ за індексом. Вектор зберігає дані у суміжному масиві і має два параметри: розмір (кількість елементів) і ємність (кількість виділених комірок пам'яті). Поки розмір не перевищує ємність, додавання нового елемента є простим записом у наступну вільну комірку. Якщо розмір більший за ємність, вектор розширює масив,

копіює всі елементи і звільняє старий масив. Вибір нового розміру визначає ефективність вектора як структури даних [1].

Стандартна стратегія розширення полягає у подвоєнні ємності. Коли масив переповнюється, виділяється новий масив удвічі більший, і всі n елементів копіюються за O(n). На перший погляд це суперечить вимозі ефективності, проте амортизований аналіз показує інше. Подвоєння відбувається після 1, 2, 4, 8, ..., n операцій, тому загальна вартість усіх копіювань за n операцій додавання становить 1 + 2 + 4 + ... + n, що дорівнює 2n. Враховуючи n операцій, отримуємо амортизовану вартість O(1) за операцію додавання. Стратегія множення на 1,5 замість 2 зменшує накладні витрати пам'яті ціною трохи частішого копіювання, але амортизована складність залишається O(1) для будь-якого постійного множника [2].

Аналіз часової складності операцій вектора дає змогу чітко розрізнити операції доступу та модифікації. Get і set за довільним індексом виконуються за O(1), оскільки безпосередньо звертаються до комірки внутрішнього масиву. Додавання в кінець виконується за амортизоване O(1) завдяки стратегії подвоєння. Натомість вставка в довільну позицію і видалення з довільної позиції вимагають зсуву всіх елементів після заданого індексу і мають складність O(n). Пошук елемента за значенням без додаткових структур потребує лінійного перегляду O(n). Цей розподіл означає, що вектор є оптимальним для задач із переважанням читання та додавання в кінець, але менш ефективним для частих вставок і видалень у середині [3].

Порівняння вектора з однозв'язним списком ілюструє основний компроміс між суміжним і розкиданим зберіганням даних у пам'яті. Список забезпечує O(1) для вставки та видалення на початку і при наявності покажчика на вузол у довільному місці, але O(n) для доступу за індексом, оскільки необхідно пройти всі попередні вузли. Вектор, навпаки, забезпечує O(1) для доступу за індексом, але O(n) для вставки і видалення в середині. Вибір між ними залежить від профілю операцій задачі: якщо переважає доступ за індексом, вектор є кращим; якщо вставки і видалення частіші в довільних місцях, список може виявитися ефективнішим.

Однією з переваг вектора над структурами на основі вузлів є локальність кешу. При доступі до пам'яті завантажується не тільки запитувана комірка, але й цілий блок, лінія кешу, зазвичай довжиною 64 байти. Оскільки елементи вектора розташовані поруч у пам'яті, доступ до одного елемента приносить кілька сусідніх комірок у кеш, і наступні доступи задовольняються з кешу без доступу до основної пам'яті. У зв'язаних структурах вузли розкидані в пам'яті, тому кожен доступ до наступного вузла, ймовірно, призведе до пропуску кешу. У великих масивах даних це може мати значний вплив на продуктивність.

У мові Java динамічний масив реалізований класом java.util.ArrayList, що є стандартною реалізацією інтерфейсу List. ArrayList зберігає елементи у внутрішньому масиві типу Object[] і подвоює його при переповненні,

починаючи з початкової ємності 10. Методи get і set виконуються за O(1), додавання в кінець за амортизоване O(1), а додавання за індексом і видалення викликають System.arraycopy і мають складність O(n). Завдяки механізму generics, ArrayList<T> зберігає елементи певного типу і забезпечує перевірку типів на етапі компіляції. Метод trimToSize дозволяє зменшити ємність до поточного розміру для економії пам'яті.

Вектор є однією з найпоширеніших структур даних у практичному програмуванні, поєднуючи зручність динамічного розміру з продуктивністю суміжного зберігання. Амортизоване O(1) для додавання, O(1) для get і set, а також висока ефективність кешу роблять його стандартним вибором для більшості задач зберігання послідовностей. Внутрішня структура вектора, зокрема стратегія подвоєння та природа амортизованого аналізу, є важливою основою для свідомого вибору між різними структурами даних залежно від характеристик конкретної задачі.

Список використаних джерел

1. Марченко О. І., Марченко О. О. Структури даних та алгоритми : підручник. Ч. 1. Київ : Просвіта, 2024. 268 с.
2. Васильєв О. М. Програмування мовою Java. Київ : Богдан, 2019. 696 с.
3. Татарчук Д. Д., Діденко Ю. В., Свєчніков Г. С. Об'єктно-орієнтоване програмування мовою Java : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2024. 153 с.

Шевченко М. І.,
студент 3 курсу ОПС «Фаховий молодший бакалавр»
спеціальності 121 «Інженерія програмного забезпечення»
освітньо-професійної програми «Інженерія програмного забезпечення»;
науковий керівник:
Савченко С. С.,
старший викладач,
циклова комісія з програмування
Фаховий коледж бізнесу та аналітики НАСОА

СТЕК ЯК ОСНОВНА СТРУКТУРА ДАНИХ: ПРИНЦИПИ ОРГАНІЗАЦІЇ ТА ОБЛАСТІ ЗАСТОСУВАННЯ

Уявімо ситуацію: функція A викликає функцію B, яка своєю чергою викликає функцію C. Кожен з цих викликів потребує збереження адреси повернення – місця, куди процесор повернеться після завершення виконання. Стек вирішує саме цю проблему, і використовується він в архітектурі процесорів ще з ранніх етапів розвитку обчислювальної техніки. Наскільки природно ця структура описує вкладені виклики, свідчить той факт, що апаратні платформи, компілятори й операційні системи покладаються на неї донині. Без розуміння принципів роботи стека неможливо повноцінно розібратися ні з рекурсією, ні з потоком виконання програми.

Стек є лінійною структурою даних з обмеженим доступом, яка працює за принципом LIFO (останній прийшов – перший вийшов), тобто елемент, доданий останнім, видаляється першим. Класичний приклад – стопка тарілок, де нова тарілка кладеться зверху і звідти ж забирається. Принцип LIFO природно моделює ситуацію, коли остання розпочата операція повинна бути завершена першою, що дуже часто трапляється в програмуванні [1].

З точки зору інтерфейсу стек дуже мінімалістичний: п'ять операцій, і більше нічого. push кладе елемент на верхівку, pop його звідти забирає і повертає, top дозволяє подивитись на верхній елемент, не чіпаючи його, isEmpty сигналізує про порожній стек, size повідомляє кількість елементів. Усі п'ять виконуються за O(1) – жодна з них не спускається глибше верхівки, тому час не залежить від того, скільки елементів зберігається нижче [2]. Саме ця гарантія сталого часу робить стек природним вибором там, де важливо керувати порядком виконання операцій.

Стек реалізується двома класичними способами. Реалізація на основі масиву зберігає елементи в суміжних комірках пам'яті, а покажчик вершини просто зростає або зменшується з push і pop. Це забезпечує відмінну ефективність кешу, оскільки дані знаходяться у суміжному блоці. Реалізація на основі зв'язаного списку додає вузол до голови при push і видаляє його при pop, що є природно динамічним, але вимагає додаткової пам'яті для покажчиків. Для

більшості задач реалізація на основі масиву з динамічним розширенням є кращим варіантом.

Найвідоміше застосування стека – стек викликів. Коли викликається функція, активний кадр з адресою повернення, параметрами та локальними змінними розміщується на ньому. Після повернення кадр видаляється, і виконання продовжується з адреси повернення. Цей механізм є повністю автоматичним і гарантує правильну рекурсію будь–якої глибини. В архітектурі x86 регістр ESP вказує на вершину стека, а інструкції PUSH і POP керують ним на рівні машинного коду.

Крім стека викликів, ця структура також використовується у багатьох інших алгоритмічних задачах. Перевірка правильного розташування дужок у виразі та перетворення інфіксної нотації в постфіксну за алгоритмом «шунтувального двору» Дейкстри базуються на стеці. Механізм Undo/Redo у текстових редакторах зберігає виконані операції на стеці скасування. Алгоритм пошуку в глибину (DFS) у своїй явній ітеративній версії також використовує стек [3].

Якщо оцінювати ефективність стека, картина однозначна. Push, pop і top звертаються лише до верхівки і не залежать від того, скільки елементів зберігається нижче, тому всі три виконуються за O(1). Очищення через clear потребує обробки всіх n елементів – звідси O(n). Пам'ять зростає пропорційно кількості збережених елементів: O(n). Для масивної реалізації варто окремо відзначити push: коли масив заповнюється вщерть і доводиться подвоїти ємність, поточне додавання обходиться дорожче, але в середньому по всіх n операціях вартість залишається сталою – сумарно 2n копіювань на n вставок дає амортизоване O(1).

У середовищі виконання Java стек присутній на багатьох рівнях. Кожен потік має стек викликів, який зберігає фрейми на методі. Коли рекурсія стає занадто глибокою, стек вичерпується, і JVM викидає виняток StackOverflowError. На рівні байт-коду JVM є стековою машиною, де більшість інструкцій зчитують операнди зі стека операндів і повертають їх назад. Знання про стек є критично важливим при дослідженні внутрішньої роботи програм на Java.

Стек і черга підкреслюють фундаментальну природу принципу доступу як класифікаційної ознаки структур даних. Черга реалізує FIFO, тобто перший доданий елемент видаляється першим, моделюючи реальні черги очікування. Стек реалізує LIFO, тобто останній доданий видаляється першим, моделюючи вкладеність. Ці два принципи діаметрально протилежні, і жоден з них не є кращим для різних видів завдань. Ця різниця є одним з перших кроків до здатності вибрати правильну структуру для конкретної проблеми.

Стек рідко буває першим, про що думають, починаючи вивчати алгоритми, і даремно: він одночасно живе в трьох різних світах – у теорії алгоритмів, у мікросхемах процесорів і в щоденному коді будь–якого

розробника. Простота інтерфейсу – лише кілька операцій – поєднується тут із концептуальною глибиною: за принципом LIFO стоїть ціла ідея вкладеності та зворотного порядку. Хто не розуміє стек, той не розуміє ні рекурсію, ні те, що відбувається всередині JVM при кожному виклику методу.

Список використаних джерел

1. Грудзинський Ю. Є. Алгоритми та структури даних : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2022. 215 с.
2. Кузьменко І. М., Дацюк О. А. Базові алгоритми та структури даних : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2022. 137 с.
3. Марченко О. І., Марченко О. О. Структури даних та алгоритми : підручник. Ч. 1. Київ : Просвіта, 2024. 268 с.

Шиловських Т. С.,
студент 3 курсу ОПС «Фаховий молодший бакалавр»
спеціальності 121 «Інженерія програмного забезпечення»
освітньо-професійної програми «Інженерія програмного забезпечення»;
науковий керівник:
Савченко С. С.,
старший викладач,
циклова комісія з програмування
Фаховий коледж бізнесу та аналітики НАСОА

ЧЕРГА З ПРІОРИТЕТАМИ ТА ДВІЙКОВА КУПА: ТЕОРЕТИЧНІ ОСНОВИ І ЗАСТОСУВАННЯ В АЛГОРИТМАХ

Дискретно-подієве моделювання є методом дослідження складних систем через симуляцію послідовності подій у часі. Симулятор дорожнього трафіку, модель мережевих протоколів або авіадиспетчерська система генерують тисячі подій, кожна з яких має часову мітку, що визначає момент її настання у модельному часі. Ключова вимога до такого симулятора полягає в тому, що події мають оброблятися у строго хронологічному порядку, незалежно від того, в якій послідовності вони надійшли до системи. Звичайна FIFO-черга тут безпорадна, адже вона зберігає порядок надходження, а не порядок пріоритету. Структура даних, що завжди надає доступ до події з найменшою часовою міткою, і є черга з пріоритетами.

Черга з пріоритетами є абстрактною структурою даних, що зберігає елементи разом з асоційованим числовим пріоритетом і завжди повертає елемент з найменшим або найбільшим значенням пріоритету незалежно від

порядку додавання. На відміну від черги FIFO, де порядок видалення визначається часом надходження, у пріоритетній черзі він визначається лише значенням пріоритету. Це фундаментально змінює семантику структури і значно розширює діапазон завдань, які вона здатна вирішити [1].

Стандартною реалізацією пріоритетної черги є бінарна купа. Купа є повним бінарним деревом, що компактно зберігається у масиві без явних покажчиків. Якщо вузол має індекс i, то лівий нащадок знаходиться за індексом 2i+1, правий за 2i+2, а батьківський вузол за індексом (i-1)/2. Властивість мінімальної купи вимагає, щоб кожен вузол був меншим або рівним своїм нащадкам, тому мінімальний елемент завжди знаходиться у корені. Операція вставки додає елемент у кінець масиву і просіює його вгору через обміни з батьківськими вузлами до відновлення властивості купи. Операція extractMin вилучає корінь, переміщує на його місце останній елемент масиву і просіює його вниз, обираючи на кожному рівні менший з двох нащадків. Обидві операції виконуються за час O(log n), оскільки висота повного бінарного дерева з n вузлами дорівнює log2(n) [2].

Алгоритм Дейкстри для знаходження найкоротшого шляху в зваженому графі є однією з класичних задач, де черга з пріоритетом є не лише зручним інструментом, але й важливим інструментом для ефективного виконання. Алгоритм має чергу вершин, кожна з яких має пріоритет, що дорівнює поточній оцінці найкоротшої відстані від джерела. На кожному кроці видаляється вершина з найменшою оцінкою, оновлюються відстані між вершиною та її сусідами, і оновлені вершини повертаються в чергу. У реалізації на основі бінарної купи загальна складність становить O((V + E) log V), де V – кількість вершин, а E – кількість ребер [3].

Іншим важливим застосуванням є алгоритм Хаффмана для безвтратного стиснення даних. Він створює оптимальне префіксне кодування символів, видаляючи вузол з найнижчою частотою з черги з пріоритетом і об'єднуючи його в новий вузол з накопиченою частотою, повертаючи його в чергу. Цей процес повторюється, поки всі вузли не будуть об'єднані в дерево Хаффмана. Черга з пріоритетами також є основою для планування завдань в операційних системах зі статичними пріоритетами. Планувальник Linux підтримує рівні nice від –20 до 19, де нижче значення означає вищий пріоритет. Він призначає процесор процесу з найвищим пріоритетом серед тих, що готові до виконання.

У Java черга з пріоритетами реалізується за допомогою java.util.PriorityQueue, яка приймає масив як двійкову купу. За замовчуванням створюється мінімальна купа, і елемент з мінімальним значенням видаляється відповідно до природного порядку. Для користувацьких типів або зворотного порядку сортування в конструктор передається Comparator, щоб критерії пріоритету могли бути визначені гнучко. Метод poll повертає і видаляє головний елемент, тоді як peek лише повертає його без видалення. Клас не є потокобезпечним; для багатопотокових сценаріїв використовується

PriorityBlockingQueue з пакету java.util.concurrent, що підтримує блокування при спробі вилучення з порожньої черги.

Черга з пріоритетами є унікальною серед структур даних як приклад того, як порядок видалення як властивість значно розширює набір завдань, які можна вирішити. Двійкова купа як її реалізація поєднує ефективність O(log n) з простотою масивного зберігання без явних покажчиків. Показово, що жодна з розглянутих задач, алгоритм Дейкстри, стиснення Хаффмана, планування процесів, не може бути ефективно вирішена без черги з пріоритетами: наївна заміна на масив із лінійним пошуком мінімуму погіршує складність до O(n) на операцію. Це підтверджує, що черга з пріоритетами є самостійною фундаментальною абстракцією, а не варіацією звичайної черги.

Список використаних джерел

1. Скулиш М. А., Суліма С. В. Алгоритми та структури даних : лабораторний практикум. Київ : КПІ ім. Ігоря Сікорського, 2021. 109 с.
2. Марченко О. І., Марченко О. О. Структури даних та алгоритми : підручник. Ч. 1. Київ : Просвіта, 2024. 268 с.
3. Кузьменко І. М., Дацюк О. А. Базові алгоритми та структури даних : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2022. 137 с.

Щербак П. Ю.,
здобувачка вищої освіти першого (бакалаврського) рівня,
спеціальність «Комп'ютерні науки»;
науковий керівник:
Ставицький О. В.,
кандидат економічних наук, доцент
завідувач кафедри інформаційних технологій,
Національна академія статистики,
обліку та аудиту

ВИКОРИСТАННЯ ХМАРНИХ ТЕХНОЛОГІЙ У ЦИФРОВІЙ ТРАНСФОРМАЦІЇ ПІДПРИЄМСТВ

У сучасних умовах глобалізації та стрімкого розвитку інформаційних технологій цифрова трансформація стала одним із ключових факторів забезпечення конкурентоспроможності підприємств. Сьогодні успіх підприємства вимірюється не багаторічною успішною діяльністю, наявністю основних активів, а здатністю змінюватися й адаптувати свій бізнес до нових умов [1]. Сьогодні організації функціонують у динамічному середовищі, де швидкість прийняття рішень, адаптивність та інноваційність відіграють вирішальну роль. Саме тому підприємства активно впроваджують цифрові інструменти, які дозволяють оптимізувати бізнес–процеси та підвищити ефективність діяльності. Трансформація підприємства є складним і багатогранним процесом, що потребує стратегічного підходу та ефективного управління [2].

Одним із найважливіших інструментів цифрової трансформації є хмарні технології. Вони забезпечують доступ до обчислювальних ресурсів через мережу Інтернет, що значно спрощує управління інформаційною інфраструктурою. Використання хмарних рішень дозволяє підприємствам швидше реагувати на зміни ринку та впроваджувати інновації з мінімальними витратами.

Хмарні технології являють собою модель надання обчислювальних ресурсів, таких як сервери, сховища даних і програмне забезпечення, через Інтернет. Це означає, що підприємствам не потрібно інвестувати значні кошти у створення та підтримку власної IT–інфраструктури, оскільки всі необхідні ресурси надаються віддалено.

Існує кілька основних моделей хмарних сервісів. Перша з них передбачає надання інфраструктури як послуги, що дозволяє підприємствам використовувати віртуальні сервери та мережі. Друга модель полягає у наданні платформи для розробки програмного забезпечення, що значно спрощує процес створення та тестування додатків. Третя модель передбачає використання готового програмного забезпечення через Інтернет без необхідності його встановлення на локальні пристрої.

Також хмарні технології можуть реалізовуватися у різних формах залежно від потреб підприємства. Вони можуть бути публічними, коли ресурси доступні широкому колу користувачів, приватними, що використовуються лише однією організацією, або гібридними, які поєднують у собі різні підходи.

Цифрова трансформація підприємств є комплексним процесом впровадження цифрових технологій у всі аспекти діяльності організації. Вона змінює не лише технічну складову бізнесу, а й підходи до управління, взаємодії з клієнтами та створення нових продуктів.

Сутність цифрової трансформації полягає у підвищенні ефективності діяльності підприємства шляхом автоматизації процесів, покращення якості обслуговування клієнтів та використання сучасних аналітичних інструментів. У цьому контексті хмарні технології виступають основою для реалізації цифрових змін, оскільки вони забезпечують необхідну гнучкість та масштабованість.

Хмарні технології відіграють важливу роль у процесі цифрової трансформації підприємств, оскільки вони дозволяють суттєво змінити підходи до організації бізнес–процесів. Однією з ключових переваг є зниження витрат, оскільки підприємства можуть відмовитися від значних капітальних інвестицій у власну інфраструктуру та перейти до моделі оплати за фактичне використання ресурсів.

Ще однією важливою характеристикою є масштабованість. Підприємства можуть швидко змінювати обсяги використання ресурсів залежно від потреб, що особливо актуально в умовах нестабільного попиту. Крім того, хмарні технології забезпечують високий рівень доступності даних, що дозволяє працівникам працювати віддалено та отримувати доступ до необхідної інформації у будь–який час.

Важливим аспектом є також прискорення впровадження інновацій. Завдяки хмарним платформам підприємства можуть швидко розробляти та тестувати нові продукти, що значно скорочує час їх виходу на ринок. Не менш значущим є і питання безпеки, оскільки сучасні провайдери хмарних послуг забезпечують високий рівень захисту даних та їх резервне копіювання.

Використання хмарних технологій надає підприємствам широкий спектр переваг. Передусім це економія ресурсів, оскільки відсутня необхідність у придбанні дорогого обладнання та його обслуговуванні. Крім того, автоматизація процесів дозволяє підвищити продуктивність праці та зменшити кількість помилок.

Хмарні рішення також сприяють покращенню аналітики даних, що дає змогу підприємствам приймати більш обґрунтовані управлінські рішення. Важливим є і те, що вони легко інтегруються з іншими цифровими інструментами, що створює єдину інформаційну екосистему.

Попри численні переваги, використання хмарних технологій пов’язане з певними труднощами. Однією з основних проблем є питання безпеки, оскільки

передача даних через Інтернет може створювати ризики їх несанкціонованого доступу. Крім того, підприємства можуть стати залежними від постачальника хмарних послуг, що ускладнює зміну платформи.

Ще одним викликом є інтеграція хмарних рішень із існуючими системами, яка може вимагати значних зусиль і ресурсів. Не менш важливими є і юридичні аспекти, пов'язані зі зберіганням даних у різних юрисдикціях.

Хмарні технології широко використовуються у різних сферах діяльності підприємств. У фінансовому секторі вони застосовуються для обробки великих обсягів даних та аналізу ризиків. У сфері торгівлі вони допомагають управляти запасами та взаємодією з клієнтами. У виробництві хмарні рішення використовуються для автоматизації процесів і контролю роботи обладнання.

Також значного поширення набули хмарні системи управління взаємовідносинами з клієнтами та системи планування ресурсів підприємства, які дозволяють інтегрувати всі бізнес–процеси в єдину систему.

У майбутньому роль хмарних технологій у діяльності підприємств буде лише зростати. Очікується подальший розвиток багатохмарних стратегій, які дозволяють використовувати послуги кількох провайдерів одночасно. Крім того, хмарні технології активно інтегруватимуться зі штучним інтелектом та іншими сучасними технологіями.

Зростатиме також рівень автоматизації процесів, що сприятиме підвищенню ефективності діяльності підприємств. Окрему увагу приділятимуть розвитку обчислень на периферії мережі, що дозволить обробляти дані ближче до джерела їх виникнення.

Отже, хмарні технології є важливим елементом цифрової трансформації підприємств і відіграють ключову роль у підвищенні їх ефективності та конкурентоспроможності. Вони забезпечують гнучкість, доступність і можливість швидкого впровадження інновацій.

Незважаючи на певні ризики, пов'язані з безпекою та залежністю від провайдерів, переваги використання хмарних технологій значно переважають можливі недоліки. Саме тому їх впровадження є необхідною умовою успішного розвитку підприємств у сучасному цифровому середовищі.

Список використаних джерел

1. Томах В. В., Сігаєва Т. Є., Мартиненко М. В. Цифрова трансформація управління підприємствами України у контексті сталого розвитку: інноваційні рішення, креативні технології. 2023. DOI: https://doi.org/10.5281/zenodo.7840221.

2. Прохорова В. В., Юхман Я. В., Янчак Ю. О. Управління трансформацією підприємств на основі цифрової когерентності // Бізнес Інформ. 2024. № 6. С. 104–111. DOI: https://doi.org/10.32983/2222–4459–2024–6–104–111.

Наукове видання

ЦИФРОВІЗАЦІЯ СУСПІЛЬСТВА ТА БІЗНЕСУ: ВИКЛИКИ ТА МОЖЛИВОСТІ ДЛЯ МОЛОДІ

VI Всеукраїнська студентська наукова конференція

Підписано до друку 03.09.2026
Формат 60x84/16. Друк офсетний
Гарнітура TimesNewRoman. Умов. друк. арк.: 3.9
Наклад прим.: 50. Замовлення №

Видавець: ТОВ «НВП «Інтерсервіс»
м. Київ, вул. Бориспільська, 9
Свідоцтво: серія ДК № 3534 від 24.07.2009 р.

Виготовлювач: ТОВ «НВП «Інтерсервіс»
м. Київ, вул. Бориспільська, 9
Свідоцтво: серія ДК № 3534 від 24.07.2009 р.