Міністерство освіти і науки України
Міжнародний гуманітарний університет
Кафедра інформаційних технологій

М.А. Мірошник

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Методичні вказівки до практичних робіт
для здобувачів вищої освіти,
які навчаються за спеціальностями
галузі «Інформаційні технології»

Одеса 2025

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

Мірошник М.А. Алгоритми та структури даних. Методичні вказівки до практичних робіт для здобувачів вищої освіти, які навчаються за спеціальностями галузі «Інформаційні технології» [Електронне видання]. Кафедра інформаційних технологій Міжнародного гуманітарного університету. Одеса, 2025. 38 с.

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

ЗМІСТ

Практична робота 1. Визначення часової складності простих алгоритмів

Мета роботи – ознайомитися з поняттям часової складності алгоритмів, навчитися оцінювати ефективність алгоритмів за допомогою асимптотичних позначень та виконувати аналіз простих алгоритмів за кількістю елементарних операцій.

Ключові положення

Ефективність алгоритму є однією з ключових характеристик програмного забезпечення, оскільки вона визначає витрати ресурсів під час виконання. Найважливішими ресурсами є час виконання та обсяг використовуваної пам'яті. Часова складність алгоритму відображає залежність кількості виконуваних операцій від розміру вхідних даних.

Для оцінювання часової складності використовуються асимптотичні позначення, які дають змогу описати поведінку алгоритму при зростанні розміру вхідних даних. Найбільш поширеними є позначення O (велике O), Ω (омега) та Θ (тета). Позначення O характеризує верхню межу складності та використовується для оцінювання найгіршого випадку. Позначення Ω визначає нижню межу, тобто найкращий випадок. Позначення Θ описує точну асимптотичну оцінку, коли верхня і нижня межі збігаються.

Аналіз алгоритму передбачає визначення кількості елементарних операцій, що виконуються під час його роботи. До таких операцій належать арифметичні дії, порівняння, присвоювання та доступ до пам'яті. У найпростішому випадку кількість операцій може бути пропорційною розміру вхідних даних, що відповідає лінійній складності $O(n)$.

Алгоритми можуть мати різну складність залежно від структури. Послідовні алгоритми, що містять один цикл, зазвичай мають лінійну складність. Вкладені цикли призводять до квадратичної складності $O(n^2)$. Якщо кожна ітерація зменшує обсяг задачі у декілька разів, виникає логарифмічна складність $O(\log n)$, яка є характерною для алгоритмів пошуку в упорядкованих структурах.

Важливим аспектом є розрізнення найгіршого, найкращого та середнього випадків виконання алгоритму. Найгірший випадок визначає максимальний час виконання і використовується для гарантування продуктивності системи. Середній випадок дає більш реалістичну оцінку, а найкращий – показує мінімальні витрати часу.

Таким чином, аналіз часової складності дозволяє порівнювати алгоритми незалежно від апаратної реалізації та обирати найбільш ефективні рішення для обробки даних.

Практичне завдання

1. Ознайомитися з поняттям часової складності алгоритмів.
2. Вивчити асимптотичні позначення O, Ω та Θ.
3. Розглянути приклади алгоритмів із різною складністю.
4. Проаналізувати алгоритм із лінійною складністю.
5. Проаналізувати алгоритм із квадратичною складністю.
6. Проаналізувати алгоритм із логарифмічною складністю.
7. Визначити кількість елементарних операцій для заданих алгоритмів.
8. Записати функцію залежності часу виконання від розміру вхідних даних.
9. Виконати спрощення отриманих виразів, відкинувши константи та другорядні члени.

10. Визначити асимптотичну складність алгоритмів.
11. Порівняти ефективність проаналізованих алгоритмів.
12. Зробити висновки щодо доцільності використання різних алгоритмів.

Методичні вказівки до виконання практичного завдання

Перед виконанням роботи необхідно засвоїти основні поняття аналізу алгоритмів, зокрема принципи оцінювання часової складності. Слід розуміти, що аналіз виконується незалежно від конкретної мови програмування та апаратного забезпечення.

На першому етапі необхідно розглянути прості алгоритми, наприклад обчислення суми елементів масиву або пошук максимального значення. Для кожного алгоритму потрібно визначити кількість виконуваних операцій у залежності від розміру вхідних даних n.

Далі слід перейти до аналізу алгоритмів із вкладеними циклами. Необхідно визначити, як змінюється кількість операцій при збільшенні n. Особливу увагу потрібно приділити випадкам, коли внутрішній цикл залежить від зовнішнього.

Під час аналізу алгоритмів із логарифмічною складністю доцільно розглянути процес поділу задачі на частини. Наприклад, у алгоритмах пошуку в упорядкованих структурах обсяг даних зменшується у два рази на кожному кроці, що визначає логарифмічний характер залежності.

Отримані вирази необхідно спростити, відкинувши сталі множники та доданки нижчого порядку. Це дозволяє зосередитися на головній тенденції зростання функції.

Після цього потрібно визначити асимптотичну складність кожного алгоритму та порівняти їх між собою. Важливо зробити висновок щодо впливу структури алгоритму на його ефективність.

Результати аналізу доцільно оформити у вигляді таблиці або структурованого опису, що містить алгоритм, функцію складності та її асимптотичну оцінку.

Контрольні питання

1. Що таке часова складність алгоритму?
2. Які ресурси враховуються при аналізі алгоритмів?
3. У чому полягає сутність позначення O?
4. Чим відрізняються позначення Ω та Θ?
5. Що таке елементарна операція?
6. Яку складність має алгоритм з одним циклом?
7. Як визначається складність алгоритму з вкладеними циклами?
8. У чому полягає сутність логарифмічної складності?
9. Чому при аналізі відкидаються константи?
10. Як порівнювати ефективність алгоритмів?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– теоретичні відомості про часову складність алгоритмів;

– аналіз обраних алгоритмів;
– визначення функцій складності;
– асимптотичні оцінки алгоритмів;
– порівняння ефективності алгоритмів;
– висновки.

Практична робота 2. Порівняння часу виконання алгоритмів для різних обсягів даних

Мета роботи – дослідити залежність часу виконання алгоритмів від розміру вхідних даних, навчитися експериментально оцінювати ефективність алгоритмів та порівнювати результати з теоретичними оцінками часової складності.

Ключові положення

Час виконання алгоритму є практичною характеристикою його ефективності, яка відображає реальні витрати обчислювальних ресурсів під час виконання програми. На відміну від теоретичної оцінки, що базується на асимптотичних залежностях, експериментальне вимірювання часу враховує особливості апаратного забезпечення, середовища виконання та реалізації алгоритму.

Залежність часу виконання від розміру вхідних даних є основним об’єктом дослідження. При збільшенні обсягу даних час виконання алгоритму зростає відповідно до його часової складності. Для алгоритмів із лінійною складністю характерне пропорційне зростання часу, для квадратичних – значно швидше, а для логарифмічних – повільніше.

Експериментальний аналіз алгоритмів передбачає проведення серії вимірювань для різних значень розміру вхідних даних. Для цього використовуються засоби вимірювання часу виконання програм, такі як вбудовані функції мови програмування або спеціалізовані бібліотеки. Важливо забезпечити однакові умови виконання для всіх експериментів, щоб результати були коректними.

Під час проведення експерименту необхідно враховувати вплив сторонніх факторів, таких як навантаження на процесор, оптимізації компілятора, кешування даних та інші особливості виконання програм. Для підвищення точності результатів доцільно виконувати багаторазові вимірювання та обчислювати середнє значення часу виконання.

Результати експериментів зазвичай подаються у вигляді таблиць або графіків, які відображають залежність часу виконання від розміру вхідних даних. Порівняння експериментальних результатів із теоретичними оцінками дозволяє підтвердити або уточнити характер складності алгоритму.

Таким чином, поєднання теоретичного та експериментального аналізу забезпечує більш повне розуміння ефективності алгоритмів та дає змогу обґрунтовано обирати оптимальні рішення для практичних задач.

Практичне завдання

1. Ознайомитися з методами експериментального вимірювання часу виконання алгоритмів.
2. Обрати декілька алгоритмів із різною часовою складністю.
3. Реалізувати обрані алгоритми засобами мови програмування.

4. Підготувати набори вхідних даних різного розміру.
5. Виконати вимірювання часу виконання алгоритмів для кожного набору даних.
6. Повторити вимірювання декілька разів для підвищення точності результатів.
7. Обчислити середній час виконання для кожного випадку.
8. Оформити результати у вигляді таблиці.
9. Побудувати графік залежності часу виконання від розміру вхідних даних.
10. Порівняти експериментальні результати з теоретичними оцінками складності.
11. Проаналізувати отримані залежності.
12. Зробити висновки щодо ефективності алгоритмів.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно обрати алгоритми для дослідження. Доцільно використовувати алгоритми з різною часовою складністю, наприклад алгоритм лінійного пошуку, алгоритм сортування простими обмінами та алгоритм бінарного пошуку.

На першому етапі необхідно реалізувати алгоритми у вибраному середовищі програмування. Слід забезпечити коректність реалізації та однакові умови виконання для всіх алгоритмів.

Далі потрібно підготувати тестові дані різного обсягу. Наприклад, можна використовувати масиви різної довжини з випадковими значеннями. Важливо, щоб дані для всіх алгоритмів були однаковими за структурою.

Для вимірювання часу виконання необхідно використати відповідні функції мови програмування. Вимірювання слід проводити декілька разів для кожного обсягу даних та обчислювати середнє значення, що дозволить зменшити вплив випадкових факторів.

Отримані результати необхідно занести до таблиці, де для кожного алгоритму вказується розмір вхідних даних та відповідний час виконання. Після цього слід побудувати графік, який відображає залежність часу виконання від розміру даних.

На основі побудованих графіків необхідно визначити характер зростання часу виконання та співвіднести його з теоретичною оцінкою складності. Особливу увагу слід приділити відмінностям між алгоритмами та їх поведінці при великих обсягах даних.

Після завершення аналізу необхідно зробити висновки щодо доцільності використання кожного алгоритму в залежності від розміру вхідних даних та вимог до продуктивності.

Контрольні питання

1. Що таке експериментальний аналіз алгоритмів?
2. Які фактори впливають на час виконання алгоритму?
3. Чому необхідно виконувати багаторазові вимірювання?
4. Як підготувати тестові дані для дослідження алгоритмів?
5. Які засоби використовуються для вимірювання часу виконання програм?
6. Як побудувати графік залежності часу виконання від розміру даних?
7. У чому полягає різниця між теоретичним та експериментальним аналізом?
8. Як визначити характер складності алгоритму за результатами експерименту?
9. Чому результати вимірювань можуть відрізнятися?
10. Як обрати ефективний алгоритм для практичного застосування?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис обраних алгоритмів;
– характеристика тестових даних;
– результати вимірювань (таблиця);
– графіки залежності часу виконання;
– аналіз отриманих результатів;
– порівняння з теоретичними оцінками;
– висновки.

Практична робота 3. Реалізація рекурсивних та ітераційних алгоритмів

Мета роботи – ознайомитися з принципами побудови рекурсивних та ітераційних алгоритмів, навчитися реалізовувати їх засобами мови програмування та порівнювати їх ефективність і особливості виконання.

Ключові положення

Рекурсія є підходом до побудови алгоритмів, при якому функція викликає саму себе для розв’язання підзадачі меншого розміру. Основною умовою коректної рекурсії є наявність базового випадку, що припиняє подальші виклики функції. Рекурсивні алгоритми природно відображають структуру задач, які мають ієрархічну або розгалужену природу, наприклад обхід дерев або обчислення факторіала.

Ітераційні алгоритми базуються на використанні циклічних конструкцій і передбачають повторення певного набору дій до досягнення заданої умови завершення. Вони не потребують додаткових викликів функцій і зазвичай є більш ефективними з точки зору використання пам’яті.

Під час виконання рекурсивних алгоритмів використовується стек викликів, у якому зберігається інформація про кожен активний виклик функції. Це може призводити до додаткових витрат пам’яті та ризику переповнення стеку при великій глибині рекурсії. Ітераційні алгоритми, як правило, не мають цього недоліку, оскільки використовують фіксовану кількість пам’яті.

Часова складність рекурсивних та ітераційних реалізацій однієї і тієї ж задачі може бути однаковою, однак фактичний час виконання може відрізнятися через накладні витрати на виклики функцій. У деяких випадках рекурсивні алгоритми можуть бути менш ефективними, якщо не застосовуються оптимізації, такі як усунення хвостової рекурсії.

Вибір між рекурсивним та ітераційним підходом залежить від структури задачі, вимог до ефективності та зручності реалізації. Рекурсія часто забезпечує більш зрозумілий і компактний код, тоді як ітерація дозволяє краще контролювати ресурси виконання.

Таким чином, аналіз та реалізація обох підходів дає змогу обґрунтовано обирати оптимальний спосіб розв’язання задачі з урахуванням її особливостей.

Практичне завдання

1. Ознайомитися з поняттям рекурсії та ітерації.
2. Розглянути приклади задач, що можуть бути розв’язані обома способами.
3. Реалізувати ітераційний алгоритм обчислення факторіала.
4. Реалізувати рекурсивний алгоритм обчислення факторіала.
5. Реалізувати ітераційний алгоритм обчислення чисел Фібоначчі.
6. Реалізувати рекурсивний алгоритм обчислення чисел Фібоначчі.
7. Порівняти результати виконання алгоритмів.
8. Виміряти час виконання рекурсивних та ітераційних реалізацій.
9. Проаналізувати використання пам’яті в обох підходах.
10. Визначити переваги та недоліки кожного підходу.
11. Зробити висновки щодо доцільності використання рекурсії та ітерації.

Методичні вказівки до виконання практичного завдання

Перед виконанням роботи необхідно чітко зрозуміти принцип роботи рекурсії. Особливу увагу слід приділити визначенню базового випадку та рекурсивного переходу, оскільки помилки на цьому етапі можуть призвести до нескінченного виконання алгоритму.

На першому етапі доцільно реалізувати ітераційні алгоритми, оскільки вони є більш очевидними та дозволяють краще зрозуміти послідовність виконання обчислень. Після цього слід реалізувати відповідні рекурсивні варіанти тих самих алгоритмів.

Під час реалізації рекурсивних алгоритмів необхідно враховувати глибину рекурсії. Для задач із великим обсягом даних це може призводити до перевищення допустимого розміру стеку. У таких випадках доцільно розглянути можливість використання ітераційного підходу.

Для порівняння ефективності необхідно виконати вимірювання часу виконання алгоритмів. Важливо забезпечити однакові умови тестування та використовувати однакові вхідні дані.

Аналіз результатів повинен включати оцінку не лише часу виконання, але й використання пам’яті. Необхідно визначити, у яких випадках рекурсивний підхід є доцільним, а коли краще застосовувати ітераційний.

Після завершення роботи необхідно сформулювати висновки щодо впливу способу реалізації алгоритму на його ефективність та ресурсоємність.

Контрольні питання

1. Що таке рекурсія?
2. Що таке базовий випадок у рекурсії?
3. У чому полягає сутність ітераційного підходу?
4. Які переваги рекурсивних алгоритмів?
5. Які недоліки рекурсії?
6. Що таке стек викликів?
7. Чим відрізняється використання пам’яті в рекурсивних та ітераційних алгоритмах?
8. У яких випадках доцільно використовувати рекурсію?
9. Як порівнюється ефективність рекурсивних та ітераційних алгоритмів?
10. Що таке хвостова рекурсія?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– теоретичні відомості про рекурсію та ітерацію;
– реалізація алгоритмів (рекурсивних та ітераційних);
– результати вимірювання часу виконання;
– аналіз використання пам’яті;
– порівняння підходів;
– висновки.

Практична робота 4. Робота з масивами та аналіз ефективності операцій

Мета роботи – ознайомитися з основними операціями над масивами, навчитися реалізовувати алгоритми обробки масивів та виконувати аналіз їх ефективності з урахуванням часової складності.

Ключові положення

Масив є однією з базових структур даних, що забезпечує зберігання послідовності елементів одного типу у безперервній області пам’яті. Така організація дозволяє здійснювати швидкий доступ до елементів за індексом, що має сталу часову складність O(1).

Основними операціями над масивами є доступ до елементів, пошук, вставлення, видалення та сортування. Доступ до елемента за індексом виконується за сталий час, тоді як пошук у невпорядкованому масиві має лінійну складність O(n). У випадку впорядкованого масиву можливе застосування більш ефективних алгоритмів, наприклад бінарного пошуку з логарифмічною складністю.

Операції вставлення та видалення в масиві пов’язані з необхідністю зміщення елементів, що призводить до лінійної складності O(n). Це обумовлено тим, що масив має фіксовану структуру та зберігається у суміжній пам’яті.

Аналіз ефективності операцій над масивами передбачає визначення кількості елементарних операцій, що виконуються під час обробки даних. Важливо враховувати як найгірший випадок, так і середній. Наприклад, пошук елемента може завершитися як на початку масиву, так і в його кінці, що впливає на загальний час виконання.

Під час роботи з масивами значну роль відіграє організація даних та вибір алгоритмів обробки. Раціональне використання масивів дозволяє забезпечити високу продуктивність програмних систем, однак у випадках частих вставлень або видалень доцільно розглянути альтернативні структури даних.

Таким чином, розуміння особливостей роботи з масивами та аналіз ефективності операцій дозволяє обирати оптимальні підходи до обробки даних у програмних системах.

Практичне завдання

1. Ознайомитися зі структурою масиву та принципами його організації в пам’яті.
2. Реалізувати створення та ініціалізацію масиву.
3. Реалізувати операцію доступу до елементів масиву за індексом.

4. Реалізувати алгоритм лінійного пошуку елемента.
5. Реалізувати алгоритм пошуку максимального та мінімального значення.
6. Реалізувати вставлення елемента у масив.
7. Реалізувати видалення елемента з масиву.
8. Визначити кількість операцій для кожної реалізованої дії.
9. Оцінити часову складність кожної операції.
10. Порівняти ефективність різних операцій над масивами.
11. Зробити висновки щодо доцільності використання масивів у різних задачах.

Методичні вказівки до виконання практичного завдання

Перед виконанням роботи необхідно розглянути принципи організації масивів у пам'яті. Слід розуміти, що всі елементи розташовані послідовно, що забезпечує швидкий доступ за індексом.

На першому етапі необхідно реалізувати базові операції роботи з масивом, зокрема створення, заповнення та виведення елементів. Це дозволяє перевірити коректність подальших реалізацій.

Далі слід реалізувати алгоритми пошуку. Для невпорядкованого масиву доцільно використати лінійний пошук, визначивши кількість операцій у найгіршому випадку.

Під час реалізації операцій вставлення та видалення необхідно враховувати зміщення елементів масиву. Слід проаналізувати, як змінюється кількість операцій залежно від позиції елемента.

Особливу увагу потрібно приділити оцінюванню часової складності. Необхідно визначити функції залежності часу виконання від розміру масиву та спростити їх до асимптотичного вигляду.

Після виконання всіх операцій необхідно порівняти їх ефективність та визначити, які операції є найбільш витратними.

Результати аналізу доцільно оформити у вигляді таблиці, що містить операцію, кількість операцій та її складність.

Контрольні питання

1. Що таке масив як структура даних?
2. Які переваги використання масивів?
3. Яку складність має доступ до елемента масиву?
4. Яку складність має лінійний пошук?
5. Чому операції вставлення та видалення є витратними?
6. Як визначається складність алгоритму пошуку?
7. У чому полягає особливість роботи з впорядкованими масивами?
8. Які фактори впливають на ефективність операцій над масивами?
9. Коли доцільно використовувати масиви?
10. Які альтернативні структури даних можуть використовуватися замість масивів?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– теоретичні відомості про масиви;
– реалізація основних операцій;
– оцінка кількості операцій;
– визначення часової складності;
– порівняння ефективності операцій;
– висновки.

Практична робота 5. Створення та обробка зв’язних списків

Мета роботи – ознайомитися з принципами побудови зв’язних списків, навчитися реалізовувати основні операції над ними та виконувати аналіз ефективності таких операцій.

Ключові положення

Зв’язний список є динамічною структурою даних, що складається з послідовності вузлів, кожен із яких містить дані та посилання на наступний елемент. На відміну від масиву, елементи зв’язного списку не розташовані у безперервній області пам’яті, що забезпечує гнучкість при зміні розміру структури.

Основною перевагою зв’язних списків є ефективність операцій вставлення та видалення елементів, які можуть виконуватися за сталий час за умови наявності посилання на відповідний вузол. Водночас доступ до елементів за позицією потребує послідовного проходження списку, що призводить до лінійної складності.

Існують різні типи зв’язних списків: однозв’язні, двозв’язні та циклічні. В однозв’язному списку кожен вузол містить посилання лише на наступний елемент. Двозв’язний список додатково містить посилання на попередній елемент, що спрощує двонапрямлений обхід. У циклічних списках останній елемент посилається на перший, утворюючи замкнену структуру.

Основними операціями над зв’язними списками є створення, додавання елементів, видалення, пошук та обхід. Під час реалізації цих операцій важливо правильно організувати роботу з посиланнями, щоб уникнути втрати даних або порушення структури списку.

Аналіз ефективності операцій над зв’язними списками базується на оцінці кількості переходів між вузлами. Наприклад, пошук елемента вимагає послідовного перегляду всіх вузлів у найгіршому випадку, що відповідає лінійній складності.

Таким чином, зв’язні списки є ефективною структурою даних для задач, що потребують частих змін структури, однак мають обмеження щодо швидкого доступу до елементів.

Практичне завдання

1. Ознайомитися з поняттям зв’язного списку.
2. Реалізувати структуру вузла зв’язного списку.
3. Реалізувати створення однозв’язного списку.

4. Реалізувати додавання елемента на початок списку.
5. Реалізувати додавання елемента в кінець списку.
6. Реалізувати вставлення елемента у довільну позицію.
7. Реалізувати видалення елемента зі списку.
8. Реалізувати пошук елемента у списку.
9. Реалізувати обхід списку та виведення його елементів.
10. Проаналізувати часову складність основних операцій.
11. Порівняти ефективність зв’язного списку з масивом.
12. Зробити висновки щодо доцільності використання зв’язних списків.

Методичні вказівки до виконання практичного завдання

Перед виконанням роботи необхідно засвоїти структуру вузла зв’язного списку. Вузол містить поле даних та посилання на наступний елемент.

На першому етапі необхідно реалізувати базову структуру однозв’язного списку. Особливу увагу слід приділити правильній ініціалізації вказівників.

Далі слід реалізувати операції додавання елементів. Для додавання на початок списку достатньо змінити посилання на перший елемент. Для додавання в кінець необхідно пройти весь список.

Під час реалізації операцій вставлення та видалення важливо правильно змінювати посилання між вузлами. Помилки на цьому етапі можуть призвести до втрати частини списку.

Для реалізації пошуку необхідно виконати послідовний обхід списку. Слід проаналізувати кількість переходів між вузлами у найгіршому випадку.

Після реалізації всіх операцій необхідно виконати аналіз їх ефективності. Доцільно порівняти отримані результати з аналогічними операціями над масивами.

Результати роботи доцільно оформити у вигляді структурованого опису або таблиці, що містить операції та їх складність.

Контрольні питання

1. Що таке зв’язний список?
2. Яка структура вузла зв’язного списку?
3. Чим зв’язний список відрізняється від масиву?
4. Які існують типи зв’язних списків?
5. Які операції можна виконувати над зв’язним списком?
6. Яку складність має пошук у зв’язному списку?
7. Які переваги зв’язних списків?
8. Які недоліки зв’язних списків?
9. У чому полягає складність операції вставлення?
10. Коли доцільно використовувати зв’язні списки?

Зміст звіту

– назва практичної роботи;
– мета виконання;

– теоретичні відомості про зв’язні списки;
– реалізація структури списку;
– реалізація основних операцій;
– аналіз ефективності операцій;
– порівняння з масивами;
– висновки.

Практична робота 6. Реалізація стеку та черги

Мета роботи – ознайомитися з принципами організації стеку та черги, навчитися реалізовувати ці структури даних засобами мови програмування та виконувати аналіз ефективності основних операцій.

Ключові положення

Стек і черга є базовими абстрактними структурами даних, що широко використовуються у програмуванні для організації обробки інформації. Вони відрізняються принципом доступу до елементів та характером виконання операцій.

Стек реалізує принцип LIFO (last in, first out), відповідно до якого останній доданий елемент обробляється першим. Основними операціями стеку є додавання елемента (push) та видалення елемента (pop). Також може використовуватися операція перегляду верхнього елемента без його видалення.

Черга реалізує принцип FIFO (first in, first out), згідно з яким перший доданий елемент обробляється першим. Основними операціями є додавання елемента в кінець (enqueue) та видалення елемента з початку (dequeue).

Стек і черга можуть бути реалізовані на основі масивів або зв’язних списків. При використанні масиву необхідно враховувати обмеження на розмір структури та можливі витрати на зміщення елементів. Використання зв’язних списків дозволяє реалізувати динамічні структури без фіксованого обсягу пам’яті.

Часова складність основних операцій для стеку та черги у типовій реалізації є сталою O(1), оскільки операції виконуються без необхідності обходу всієї структури. Це робить їх ефективними для задач, що потребують швидкого додавання та видалення елементів.

Стек широко використовується при реалізації рекурсії, обробці виразів та організації викликів функцій. Черга застосовується у задачах планування, обробки подій та алгоритмах обходу графів.

Таким чином, стек і черга є фундаментальними структурами даних, розуміння яких є необхідним для ефективної розробки програмного забезпечення.

Практичне завдання

1. Ознайомитися з принципами роботи стеку та черги.
2. Реалізувати стек на основі масиву.
3. Реалізувати операції push, pop та перегляду верхнього елемента.
4. Реалізувати чергу на основі масиву.
5. Реалізувати операції enqueue та dequeue.
6. Реалізувати стек на основі зв’язного списку.

7. Реалізувати чергу на основі зв’язного списку.
8. Перевірити коректність роботи реалізованих структур.
9. Визначити часову складність основних операцій.
10. Порівняти різні способи реалізації стеку та черги.
11. Зробити висновки щодо ефективності реалізацій.

Методичні вказівки до виконання практичного завдання

Перед виконанням роботи необхідно розглянути принципи роботи стеку та черги. Особливу увагу слід приділити різниці між підходами LIFO та FIFO.

На першому етапі доцільно реалізувати стек на основі масиву. Необхідно визначити змінну, що вказує на вершину стеку, та забезпечити коректну обробку ситуацій переповнення та порожнього стеку.

Далі слід реалізувати чергу. У випадку масивної реалізації необхідно враховувати можливість використання циклічного буфера для уникнення неефективного зміщення елементів.

Після цього необхідно реалізувати стек і чергу на основі зв’язних списків. Це дозволяє усунути обмеження на розмір структури та забезпечити більш гнучке управління пам’яттю.

Особливу увагу слід приділити перевірці коректності роботи операцій. Необхідно протестувати додавання та видалення елементів у різних сценаріях.

Після реалізації структур необхідно виконати аналіз їх ефективності. Слід визначити часову складність операцій та порівняти різні підходи до реалізації.

У завершенні роботи необхідно зробити висновки щодо переваг та недоліків різних реалізацій стеку та черги.

Контрольні питання

1. Що таке стек?
2. У чому полягає принцип LIFO?
3. Які основні операції стеку?
4. Що таке черга?
5. У чому полягає принцип FIFO?
6. Які основні операції черги?
7. Які способи реалізації стеку існують?
8. Які способи реалізації черги існують?
9. Яку складність мають основні операції стеку та черги?
10. У яких задачах використовуються стек і черга?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– теоретичні відомості про стек і чергу;
– реалізація структур даних;
– опис основних операцій;

– аналіз ефективності;
– порівняння реалізацій;
– висновки.

Практична робота 7. Реалізація лінійного та бінарного пошуку

Мета роботи – ознайомитися з алгоритмами лінійного та бінарного пошуку, навчитися реалізовувати їх засобами мови програмування, порівнювати ефективність цих алгоритмів та визначати умови доцільного використання кожного з них.

Ключові положення

Пошук елемента є однією з базових задач обробки даних, яка полягає у знаходженні елемента з заданим значенням у структурі даних. Ефективність алгоритму пошуку має важливе значення, оскільки вона безпосередньо впливає на швидкодію програмної системи.

Лінійний пошук є найпростішим алгоритмом пошуку. Його сутність полягає у послідовному перегляді елементів масиву від початку до кінця до моменту знаходження потрібного значення або завершення перегляду всієї послідовності. Цей алгоритм не потребує попереднього впорядкування даних і може застосовуватися до будь-яких послідовних структур. Його часова складність у найгіршому випадку є лінійною і становить O(n).

Бінарний пошук є більш ефективним алгоритмом, проте його застосування можливе лише для впорядкованих масивів або інших структур із впорядкованими даними. Алгоритм ґрунтується на послідовному поділі області пошуку навпіл. На кожному етапі порівнюється шукане значення із середнім елементом. Якщо значення менше за середній елемент, пошук продовжується у лівій частині масиву, якщо більше – у правій. Такий підхід забезпечує логарифмічну часову складність O(log n).

Порівняння лінійного та бінарного пошуку показує, що вибір алгоритму залежить не лише від кількості елементів, але й від організації даних. Якщо масив не впорядкований і не передбачається попереднє сортування, доцільним є застосування лінійного пошуку. Якщо ж дані впорядковані або можуть бути впорядковані заздалегідь, бінарний пошук забезпечує значно вищу ефективність.

Під час аналізу алгоритмів пошуку важливо враховувати найкращий, середній та найгірший випадки виконання. Для лінійного пошуку найкращий випадок реалізується, коли шуканий елемент розташований на початку масиву. Для бінарного пошуку найкращий випадок має місце, коли шуканий елемент одразу опиняється в середині масиву.

Таким чином, лінійний та бінарний пошук є важливими базовими алгоритмами, знання яких дозволяє обґрунтовано обирати спосіб пошуку залежно від структури даних та вимог до продуктивності.

Практичне завдання

1. Ознайомитися з поняттям пошуку в масивах.
2. Реалізувати алгоритм лінійного пошуку.
3. Перевірити роботу алгоритму на невпорядкованому масиві.
4. Реалізувати алгоритм бінарного пошуку.

5. Підготувати впорядкований масив для виконання бінарного пошуку.
6. Перевірити роботу алгоритму бінарного пошуку на різних наборах даних.
7. Визначити кількість порівнянь у лінійному пошуку.
8. Визначити кількість порівнянь у бінарному пошуку.
9. Порівняти ефективність обох алгоритмів для різних розмірів масивів.
10. Проаналізувати умови використання кожного алгоритму.
11. Зробити висновки щодо доцільності застосування лінійного та бінарного пошуку.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно засвоїти відмінності між послідовним і подільним підходами до пошуку. Лінійний пошук перевіряє елементи один за одним, тоді як бінарний пошук щоразу зменшує область пошуку вдвічі.

На першому етапі необхідно реалізувати алгоритм лінійного пошуку для масиву довільних даних. Під час реалізації слід передбачити два можливі результати: елемент знайдено або елемент відсутній.

Далі потрібно реалізувати алгоритм бінарного пошуку. Перед його застосуванням слід переконатися, що масив є впорядкованим за зростанням або спаданням. Особливу увагу потрібно приділити правильному обчисленню середнього індексу та оновленню меж області пошуку.

Після реалізації обох алгоритмів необхідно виконати серію тестів на масивах різного розміру. Доцільно перевірити випадки, коли елемент знаходиться на початку, у середині, у кінці масиву, а також коли шуканий елемент відсутній.

Під час аналізу результатів слід визначити кількість порівнянь, які виконує кожен алгоритм. Це дозволить більш наочно оцінити їх ефективність.

Особливу увагу необхідно приділити співвідношенню між витратами на попереднє впорядкування даних та виграшем від використання бінарного пошуку. У невеликих масивах або при одноразовому пошуку лінійний підхід може бути цілком доцільним.

Після завершення роботи необхідно зробити висновки щодо переваг, недоліків та сфер доцільного застосування обох алгоритмів.

Контрольні питання

1. Що таке алгоритм пошуку?
2. У чому полягає сутність лінійного пошуку?
3. Які переваги лінійного пошуку?
4. Які недоліки лінійного пошуку?
5. У чому полягає сутність бінарного пошуку?
6. Яка основна умова застосування бінарного пошуку?
7. Яку часову складність має лінійний пошук?
8. Яку часову складність має бінарний пошук?
9. Чому бінарний пошук є ефективнішим за лінійний?
10. У яких випадках доцільно використовувати лінійний пошук?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– теоретичні відомості про алгоритми пошуку;
– реалізація лінійного пошуку;
– реалізація бінарного пошуку;
– результати тестування алгоритмів;
– порівняння кількості порівнянь і часової складності;
– висновки.

Практична робота 8. Реалізація простих алгоритмів сортування

Мета роботи – ознайомитися з принципами роботи простих алгоритмів сортування, навчитися реалізовувати їх засобами мови програмування та виконувати аналіз їх ефективності.

Ключові положення

Сортування є однією з базових операцій обробки даних, що полягає у впорядкуванні елементів за певним критерієм, найчастіше за зростанням або спаданням. Ефективність алгоритмів сортування має суттєвий вплив на продуктивність програмних систем, особливо при роботі з великими обсягами даних.

До простих алгоритмів сортування належать сортування обміном, сортування вибором та сортування вставками. Вони є інтуїтивно зрозумілими та зручними для реалізації, однак мають відносно невисоку ефективність у порівнянні з більш складними алгоритмами.

Сортування обміном базується на багаторазовому порівнянні сусідніх елементів та їх перестановці у разі неправильного порядку. У результаті найбільші або найменші елементи поступово переміщуються у відповідну частину масиву.

Сортування вибором полягає у знаходженні мінімального або максимального елемента та його розміщенні у відповідній позиції. Цей процес повторюється для решти елементів масиву.

Сортування вставками передбачає поступове формування впорядкованої частини масиву шляхом вставлення кожного нового елемента у відповідну позицію. Цей алгоритм є ефективнішим для майже впорядкованих даних.

Часова складність більшості простих алгоритмів сортування у найгіршому випадку є квадратичною $O(n^2)$, що обмежує їх застосування для великих масивів. Проте для невеликих обсягів даних або у випадках, коли важлива простота реалізації, вони можуть бути цілком доцільними.

Аналіз алгоритмів сортування передбачає визначення кількості порівнянь і перестановок, що виконуються під час їх роботи. Це дозволяє оцінити ефективність алгоритму та порівняти його з іншими методами сортування.

Таким чином, прості алгоритми сортування є важливим етапом вивчення методів обробки даних та формують основу для розуміння більш складних алгоритмів.

Практичне завдання

1. Ознайомитися з поняттям сортування даних.
2. Реалізувати алгоритм сортування обміном.
3. Реалізувати алгоритм сортування вибором.
4. Реалізувати алгоритм сортування вставками.
5. Перевірити роботу алгоритмів на різних наборах даних.
6. Визначити кількість порівнянь для кожного алгоритму.
7. Визначити кількість перестановок для кожного алгоритму.
8. Оцінити часову складність алгоритмів.
9. Порівняти ефективність алгоритмів сортування.
10. Проаналізувати вплив початкового стану масиву на ефективність сортування.
11. Зробити висновки щодо доцільності використання кожного алгоритму.

Методичні вказівки до виконання практичного завдання

Перед виконанням роботи необхідно ознайомитися з принципами впорядкування даних. Слід розуміти, що різні алгоритми сортування мають різні характеристики ефективності залежно від обсягу та структури даних.

На першому етапі необхідно реалізувати алгоритм сортування обміном. Слід організувати вкладені цикли, що забезпечують порівняння сусідніх елементів та їх перестановку.

Далі потрібно реалізувати сортування вибором. Для цього необхідно на кожному етапі знаходити мінімальний елемент у невпорядкованій частині масиву та переміщувати його на відповідну позицію.

Після цього слід реалізувати сортування вставками. Особливу увагу необхідно приділити процесу зміщення елементів для вставлення поточного елемента у правильну позицію.

Після реалізації алгоритмів необхідно протестувати їх на масивах різного розміру та з різною початковою впорядкованістю. Доцільно розглянути випадки випадкових, впорядкованих та обернено впорядкованих масивів.

Під час аналізу результатів необхідно визначити кількість порівнянь і перестановок. Це дозволить оцінити ефективність алгоритмів більш точно.

Після завершення аналізу необхідно зробити висновки щодо переваг і недоліків кожного алгоритму та визначити сфери їх доцільного застосування.

Контрольні питання

1. Що таке сортування даних?
2. Які основні алгоритми простого сортування?
3. У чому полягає принцип сортування обміном?
4. У чому полягає принцип сортування вибором?
5. У чому полягає принцип сортування вставками?
6. Яку часову складність мають прості алгоритми сортування?
7. Які фактори впливають на ефективність сортування?
8. Чому сортування вставками є ефективним для частково впорядкованих даних?

9. Як оцінити кількість порівнянь і перестановок?
10. У яких випадках доцільно використовувати прості алгоритми сортування?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– теоретичні відомості про алгоритми сортування;
– реалізація алгоритмів сортування;
– результати тестування;
– аналіз кількості порівнянь і перестановок;
– оцінка часової складності;
– висновки.

Практична робота 9. Реалізація ефективних алгоритмів сортування

Мета роботи – ознайомитися з принципами роботи ефективних алгоритмів сортування, навчитися реалізовувати їх засобами мови програмування та виконувати аналіз їх ефективності у порівнянні з простими методами сортування.

Ключові положення

Ефективні алгоритми сортування застосовуються у випадках, коли обсяг даних є значним і використання простих квадратичних методів стає недоцільним. На відміну від сортування обміном, вибором або вставками, ефективні алгоритми забезпечують менше зростання часу виконання при збільшенні кількості елементів. У більшості випадків їх часова складність становить O(n log n), що робить їх придатними для практичного використання у програмних системах.

До найпоширеніших ефективних алгоритмів належать швидке сортування, сортування злиттям та пірамідальне сортування. Кожен із цих алгоритмів має власний принцип організації обробки даних, переваги та обмеження.

Швидке сортування базується на виборі опорного елемента та поділі масиву на частини відносно цього елемента. Після цього до кожної частини рекурсивно застосовується той самий підхід. У середньому цей алгоритм працює дуже ефективно, хоча в найгіршому випадку його складність може зростати до $O(n^2)$.

Сортування злиттям ґрунтується на принципі поділу масиву на дві частини, окремого сортування кожної частини та подальшого злиття впорядкованих підмасивів. Цей алгоритм забезпечує стабільну часову складність O(n log n), однак потребує додаткової пам’яті для виконання операції злиття.

Пірамідальне сортування використовує структуру даних купа. Спочатку масив перетворюється на бінарну купу, після чого найбільший або найменший елемент послідовно переміщується у кінець масиву. Цей алгоритм не потребує значного додаткового обсягу пам’яті та має складність O(n log n).

Під час аналізу алгоритмів сортування важливо враховувати не лише часову складність, але й потребу у додатковій пам’яті, стабільність сортування, чутливість до початкового стану даних та особливості реалізації.

Таким чином, ефективні алгоритми сортування є важливим інструментом обробки великих обсягів даних і дозволяють забезпечити вищу продуктивність програмних систем у порівнянні з простими методами сортування.

Практичне завдання

1. Ознайомитися з поняттям ефективного сортування.
2. Реалізувати алгоритм швидкого сортування.
3. Реалізувати алгоритм сортування злиттям.
4. Реалізувати алгоритм пірамідального сортування.
5. Перевірити роботу алгоритмів на різних наборах даних.
6. Порівняти результати сортування для випадкових, впорядкованих та обернено впорядкованих масивів.
7. Визначити часову складність кожного алгоритму.
8. Оцінити потребу алгоритмів у додатковій пам’яті.
9. Порівняти ефективні алгоритми з простими алгоритмами сортування.
10. Визначити переваги та недоліки кожного алгоритму.
11. Зробити висновки щодо доцільності використання різних методів сортування.

Методичні вказівки до виконання практичного завдання

Перед виконанням роботи необхідно ознайомитися з особливостями алгоритмів сортування, що мають складність O(n log n). Слід розуміти, що підвищення ефективності алгоритму зазвичай супроводжується ускладненням його структури порівняно з простими методами сортування.

На першому етапі доцільно реалізувати швидке сортування. Особливу увагу слід приділити вибору опорного елемента та процедурі розподілу елементів на дві частини. Необхідно перевірити правильність рекурсивної обробки підмасивів.

Далі потрібно реалізувати сортування злиттям. Для цього необхідно коректно організувати поділ масиву на частини та процедуру злиття двох впорядкованих підмасивів в один. Особливу увагу слід приділити використанню допоміжної пам’яті.

Після цього необхідно реалізувати пірамідальне сортування. Слід побудувати бінарну купу та забезпечити правильне відновлення її властивостей після кожного переміщення кореневого елемента.

Після реалізації алгоритмів необхідно перевірити їх роботу на масивах різного розміру та різної структури. Доцільно використати однакові набори даних для всіх алгоритмів, щоб забезпечити коректність порівняння.

Під час аналізу результатів потрібно визначити не лише правильність сортування, але й особливості роботи алгоритмів у різних умовах. Необхідно звернути увагу на швидкість виконання, обсяг додаткової пам’яті та складність програмної реалізації.

Після завершення роботи необхідно зробити висновки щодо того, який алгоритм є більш доцільним залежно від обсягу даних, вимог до швидкодії та обмежень на використання пам’яті.

Контрольні питання

1. Які алгоритми належать до ефективних методів сортування?
2. У чому полягає принцип швидкого сортування?
3. Яка середня часова складність швидкого сортування?
4. У чому полягає принцип сортування злиттям?
5. Чому сортування злиттям потребує додаткової пам’яті?
6. У чому полягає принцип пірамідального сортування?
7. Яку часову складність мають ефективні алгоритми сортування?
8. Чим ефективні алгоритми відрізняються від простих?
9. Які фактори впливають на вибір алгоритму сортування?
10. У яких випадках доцільно використовувати ефективні алгоритми сортування?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– теоретичні відомості про ефективні алгоритми сортування;
– реалізація алгоритмів;
– результати тестування;
– порівняння алгоритмів;
– аналіз часової складності та використання пам’яті;
– висновки.

Практична робота 10. Створення бінарного дерева та його обхід

Мета роботи – ознайомитися з принципами побудови бінарного дерева, навчитися реалізовувати структуру вузла, створювати дерево та виконувати основні способи його обходу.

Ключові положення

Бінарне дерево є ієрархічною структурою даних, у якій кожен вузол може мати не більше двох нащадків. Зазвичай їх називають лівим і правим піддеревом. Така структура широко використовується у програмуванні для подання впорядкованих даних, організації пошуку, побудови дерев виразів, синтаксичного аналізу та багатьох інших задач.

Основними елементами бінарного дерева є корінь, вузли, листки, батьківські вузли, дочірні вузли, рівні та висота дерева. Коренем називається верхній вузол дерева, від якого починається доступ до всіх інших елементів. Вузли, що не мають нащадків, називаються листками.

Бінарне дерево може бути порожнім або містити один чи більше вузлів. Кожен вузол, окрім значення даних, зазвичай містить посилання на лівого та правого нащадка. Саме через ці посилання забезпечується зв’язок між елементами дерева.

Однією з найважливіших операцій при роботі з бінарними деревами є обхід. Обхід дерева означає послідовне відвідування всіх його вузлів за певним правилом. Найпоширенішими є три способи обходу: префіксний, симетричний та постфіксний.

Префіксний обхід передбачає спочатку обробку поточного вузла, потім обхід лівого піддерева і після цього обхід правого піддерева. Такий підхід використовується, зокрема, для копіювання дерева або побудови префіксного запису виразів.

Симетричний обхід виконується у такому порядку: спочатку ліве піддерево, потім поточний вузол, а далі праве піддерево. У випадку бінарного дерева пошуку саме цей спосіб обходу дозволяє отримати елементи у впорядкованому вигляді.

Постфіксний обхід передбачає спочатку обхід лівого піддерева, потім правого, а вже після цього обробку поточного вузла. Такий підхід часто використовується при видаленні дерева або обчисленні значень у дереві виразів.

Ефективність обходу дерева визначається кількістю вузлів. Кожен вузол відвідується один раз, тому часова складність усіх основних способів обходу становить O(n), де n – кількість вузлів у дереві.

Таким чином, бінарне дерево є важливою структурою даних, а реалізація його обходу формує основу для подальшого вивчення дерев пошуку, куп та інших ієрархічних структур.

Практичне завдання

1. Ознайомитися з поняттям бінарного дерева як структури даних.
2. Реалізувати структуру вузла бінарного дерева.
3. Створити порожнє бінарне дерево.
4. Реалізувати додавання вузлів до дерева за заданою схемою.
5. Побудувати приклад бінарного дерева з декількох вузлів.
6. Реалізувати префіксний обхід дерева.
7. Реалізувати симетричний обхід дерева.
8. Реалізувати постфіксний обхід дерева.
9. Вивести результати кожного способу обходу.
10. Порівняти порядок відвідування вузлів для різних способів обходу.
11. Оцінити часову складність операцій обходу.
12. Зробити висновки щодо особливостей подання та обходу бінарного дерева.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно засвоїти основні поняття, пов’язані з деревами, зокрема корінь, листок, піддерево та висоту дерева. Слід розуміти, що бінарне дерево є рекурсивною структурою, оскільки кожне його піддерево також є бінарним деревом.

На першому етапі необхідно реалізувати структуру вузла. Вона повинна містити поле для збереження значення та два посилання на лівого і правого нащадка. Після цього потрібно створити дерево та вручну або програмним шляхом додати до нього декілька вузлів.

Далі слід реалізувати основні способи обходу дерева. Доцільно почати з рекурсивного підходу, оскільки він природно відповідає структурі дерева. Для кожного способу обходу потрібно чітко дотримуватися порядку відвідування вузлів.

Після реалізації обходів необхідно перевірити правильність роботи програми на конкретному прикладі дерева. Для цього слід зафіксувати структуру дерева та порівняти отриманий порядок обходу з очікуваним результатом.

Особливу увагу потрібно приділити відмінностям між трьома способами обходу. Хоча всі вони відвідують ті самі вузли, порядок їх обробки буде різним, що визначає сферу застосування кожного підходу.

Після цього необхідно оцінити ефективність реалізованих алгоритмів. Оскільки кожен вузол відвідується один раз, слід показати, що всі способи обходу мають лінійну часову складність.

На завершальному етапі потрібно зробити висновки щодо особливостей використання бінарних дерев та значення обходів для подальшої роботи з ієрархічними структурами даних.

Контрольні питання

1. Що таке бінарне дерево?
2. Які основні елементи бінарного дерева?
3. Що таке корінь дерева?
4. Які вузли називаються листками?
5. У чому полягає сутність префіксного обходу?
6. У чому полягає сутність симетричного обходу?
7. У чому полягає сутність постфіксного обходу?
8. Яка часова складність обходу бінарного дерева?
9. Чому рекурсія є зручною для реалізації обходу дерева?
10. У яких задачах використовуються обходи бінарного дерева?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– теоретичні відомості про бінарні дерева;
– реалізація структури вузла та побудови дерева;
– реалізація префіксного, симетричного та постфіксного обходу;
– результати тестування;
– аналіз особливостей обходу;
– висновки.

Практична робота 11. Реалізація операцій у бінарному дереві пошуку

Мета роботи – ознайомитися з принципами побудови бінарного дерева пошуку, навчитися реалізовувати основні операції над цією структурою даних та виконувати аналіз їх ефективності.

Ключові положення

Бінарне дерево пошуку є ієрархічною структурою даних, у якій кожен вузол містить ключ та два посилання на дочірні вузли. Основною властивістю цієї структури є те, що всі ключі в лівому піддереві є меншими за ключ поточного вузла, а всі ключі в правому піддереві є більшими. Така організація забезпечує ефективне виконання операцій пошуку, вставлення та видалення елементів.

Пошук у бінарному дереві пошуку виконується шляхом послідовного порівняння шуканого ключа зі значенням поточного вузла. Якщо ключ менший, перехід здійснюється до лівого піддерева, якщо більший – до правого. Процес повторюється до знаходження потрібного елемента або досягнення порожнього посилання.

Операція вставлення нового елемента реалізується за аналогічним принципом. Новий вузол додається у таку позицію, яка не порушує властивість впорядкованості дерева.

Видалення вузла є більш складною операцією, оскільки потребує збереження структури дерева пошуку. Якщо вузол не має нащадків, він просто видаляється. Якщо вузол має одного нащадка, його місце займає цей нащадок. Якщо вузол має двох нащадків, зазвичай використовується заміна значення на мінімальний елемент правого піддерева або максимальний елемент лівого піддерева з подальшим видаленням відповідного вузла.

Ефективність операцій у бінарному дереві пошуку залежить від його висоти. У випадку збалансованого дерева пошук, вставлення та видалення мають логарифмічну складність O(log n). Якщо ж дерево вироджується у лінійну структуру, складність цих операцій зростає до O(n).

Окрім основних операцій, важливе значення мають способи обходу дерева. До них належать симетричний, префіксний та постфіксний обходи. Симетричний обхід дозволяє отримати елементи у впорядкованому вигляді.

Таким чином, бінарне дерево пошуку є важливою структурою даних для організації ефективного зберігання та пошуку інформації, однак його ефективність суттєво залежить від форми дерева.

Практичне завдання

1. Ознайомитися зі структурою бінарного дерева пошуку.
2. Реалізувати структуру вузла дерева.
3. Реалізувати створення порожнього дерева.
4. Реалізувати операцію вставлення вузла у дерево.
5. Реалізувати операцію пошуку елемента у дереві.
6. Реалізувати операцію видалення вузла з дерева.
7. Реалізувати симетричний обхід дерева.
8. Реалізувати префіксний обхід дерева.
9. Реалізувати постфіксний обхід дерева.
10. Перевірити коректність роботи реалізованих операцій.
11. Оцінити часову складність основних операцій.
12. Зробити висновки щодо ефективності бінарного дерева пошуку.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно засвоїти основну властивість бінарного дерева пошуку, відповідно до якої значення лівого піддерева є меншими, а правого – більшими за значення поточного вузла. Саме ця властивість забезпечує ефективність основних операцій.

На першому етапі необхідно реалізувати структуру вузла дерева. Вузол повинен містити поле ключа та посилання на лівий і правий дочірні вузли.

Далі слід реалізувати операцію вставлення. Для цього необхідно організувати рекурсивний або ітераційний перехід по дереву до знаходження правильної позиції для нового вузла.

Після цього потрібно реалізувати пошук елемента. Слід перевірити випадки, коли елемент знаходиться у корені дерева, в одному з піддерев або відсутній у структурі.

Особливу увагу необхідно приділити реалізації видалення. Доцільно окремо перевірити три випадки: видалення листка, видалення вузла з одним нащадком та видалення вузла з двома нащадками.

Після реалізації основних операцій необхідно реалізувати різні способи обходу дерева. Симетричний обхід слід використати для перевірки впорядкованості елементів.

На завершальному етапі потрібно виконати аналіз ефективності реалізованих операцій та визначити вплив висоти дерева на складність пошуку, вставлення і видалення.

Контрольні питання

1. Що таке бінарне дерево пошуку?
2. Яка основна властивість бінарного дерева пошуку?
3. Як виконується пошук у бінарному дереві пошуку?
4. Як виконується вставлення нового вузла?
5. Які випадки потрібно враховувати при видаленні вузла?
6. Що таке симетричний обхід дерева?
7. Яке призначення префіксного обходу?
8. Яке призначення постфіксного обходу?
9. Від чого залежить ефективність операцій у дереві?
10. У чому полягає проблема виродження дерева?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– теоретичні відомості про бінарне дерево пошуку;
– реалізація структури вузла;
– реалізація операцій вставлення, пошуку та видалення;
– реалізація обходів дерева;
– аналіз часової складності;
– висновки.

Практична робота 12. Реалізація хеш-таблиці

Мета роботи – ознайомитися з принципами побудови хеш-таблиць, навчитися реалізовувати основні операції над цією структурою даних та виконувати аналіз їх ефективності.

Ключові положення

Хеш-таблиця є структурою даних, що призначена для швидкого зберігання, пошуку та видалення елементів за ключем. Основу її роботи становить хеш-функція, яка перетворює ключ на числовий індекс комірки таблиці. Завдяки цьому доступ до елемента у середньому випадку може виконуватися за сталий час.

Хеш-функція повинна забезпечувати рівномірний розподіл ключів по комірках таблиці. Якщо різні ключі відображаються в одну й ту саму комірку, виникає колізія. Колізії є природним явищем при роботі хеш-таблиць, тому необхідно передбачати способи їх розв'язання.

Основними методами розв'язання колізій є метод ланцюжків та відкрита адресація. У методі ланцюжків кожна комірка таблиці містить список елементів, що мають однакове хеш-значення. У методах відкритої адресації у разі колізії виконується пошук іншої вільної позиції за певним правилом, наприклад лінійним або квадратичним пробуванням.

Основними операціями над хеш-таблицею є вставлення елемента, пошук за ключем та видалення. У середньому випадку ці операції мають часову складність O(1), однак у найгіршому випадку, коли кількість колізій значна, складність може зростати до O(n).

Ефективність хеш-таблиці залежить від якості хеш-функції, розміру таблиці та коефіцієнта заповнення. При надмірному заповненні таблиці кількість колізій зростає, що погіршує швидкодію. У таких випадках може використовуватися перехешування, тобто збільшення розміру таблиці та повторний розподіл усіх елементів.

Хеш-таблиці широко застосовуються у системах швидкого пошуку, словниках, кешах, механізмах індексації та інших компонентах програмних систем, де потрібен швидкий доступ до даних за ключем.

Таким чином, хеш-таблиця є однією з найефективніших структур даних для задач асоціативного доступу, однак її результативність визначається правильним вибором способу хешування та організації обробки колізій.

Практичне завдання

1. Ознайомитися з принципом роботи хеш-таблиці.
2. Реалізувати просту хеш-функцію для числових або рядкових ключів.
3. Реалізувати структуру хеш-таблиці.
4. Реалізувати операцію вставлення елемента у хеш-таблицю.
5. Реалізувати операцію пошуку елемента за ключем.
6. Реалізувати операцію видалення елемента за ключем.
7. Реалізувати один зі способів розв'язання колізій.
8. Перевірити роботу хеш-таблиці на наборі тестових даних.
9. Проаналізувати випадки виникнення колізій.
10. Оцінити часову складність основних операцій.
11. Зробити висновки щодо ефективності хеш-таблиці.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно засвоїти, що хеш-таблиця не зберігає елементи у впорядкованому вигляді, а використовує обчислений індекс для швидкого доступу до них. Саме тому центральним елементом цієї структури є хеш-функція.

На першому етапі необхідно реалізувати просту хеш-функцію. Для числових ключів можна використати операцію остачі від ділення на розмір таблиці. Для рядкових ключів доцільно використати посимвольне обчислення числового коду з подальшим приведенням до допустимого діапазону індексів.

Далі потрібно створити структуру хеш-таблиці, яка міститиме масив комірок. Після цього слід реалізувати операцію вставлення елемента. Якщо обрана комірка вже зайнята, необхідно застосувати механізм розв’язання колізії.

Під час реалізації пошуку слід враховувати, що елемент може знаходитися не лише у початково обчисленій позиції, а й у пов’язаній структурі або в іншій позиції, знайденій методом пробування.

Операцію видалення також необхідно реалізувати з урахуванням особливостей обраного способу обробки колізій. У випадку відкритої адресації слід уникати порушення послідовності пошуку.

Після реалізації основних операцій необхідно протестувати хеш-таблицю на різних ключах та проаналізувати, у яких випадках виникають колізії. Доцільно оцінити вплив розміру таблиці на кількість конфліктів.

На завершальному етапі необхідно визначити часову складність операцій вставлення, пошуку та видалення, а також сформулювати висновки щодо переваг і обмежень хеш-таблиць.

Контрольні питання

1. Що таке хеш-таблиця?
2. Яке призначення хеш-функції?
3. Що таке колізія?
4. Які існують способи розв’язання колізій?
5. У чому полягає метод ланцюжків?
6. У чому полягає відкрита адресація?
7. Яку часову складність мають основні операції хеш-таблиці у середньому випадку?
8. Від чого залежить ефективність хеш-таблиці?
9. Що таке коефіцієнт заповнення таблиці?
10. У яких задачах доцільно використовувати хеш-таблиці?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– теоретичні відомості про хеш-таблиці;
– опис реалізованої хеш-функції;
– реалізація основних операцій;
– опис способу розв’язання колізій;
– результати тестування;
– аналіз ефективності;
– висновки.

Практична робота 13. Подання графа та реалізація його обходу

Мета роботи – ознайомитися зі способами подання графів у програмуванні, навчитися реалізовувати основні форми їх зберігання та виконувати обхід графа в глибину і в ширину.

Ключові положення

Граф є структурою даних, що використовується для подання множини вершин та зв'язків між ними. Вершини графа відповідають окремим об'єктам, а ребра визначають наявність зв'язків або відношень між цими об'єктами. Графи широко застосовуються для моделювання комп'ютерних мереж, транспортних систем, соціальних зв'язків, залежностей між модулями програмного забезпечення та багатьох інших об'єктів.

Графи поділяються на орієнтовані та неорієнтовані. В орієнтованому графі ребра мають напрям, тобто визначають перехід від однієї вершини до іншої. У неорієнтованому графі зв'язок між вершинами є двостороннім. Крім того, графи можуть бути зваженими або незваженими залежно від того, чи мають ребра числові характеристики.

Для програмного подання графів найчастіше використовуються матриця суміжності та список суміжності. Матриця суміжності є двовимірною структурою, у якій елемент на перетині рядка і стовпця вказує на наявність або відсутність ребра між відповідними вершинами. Такий спосіб є зручним для щільних графів і забезпечує швидку перевірку існування ребра. Список суміжності подає для кожної вершини перелік суміжних вершин. Це подання є більш економним за використанням пам'яті у розріджених графах.

Обхід графа є однією з базових операцій, що дозволяє послідовно відвідати всі вершини, досяжні з певної початкової вершини. Найпоширенішими алгоритмами обходу є пошук у глибину та пошук у ширину.

Пошук у глибину базується на переході від поточної вершини до однієї з суміжних, після чого процес продовжується вглиб графа до тих пір, поки існує можливість переходу. Після досягнення вершини, з якої немає невідвіданих суміжних вершин, виконується повернення до попередніх вершин. Такий алгоритм природно реалізується рекурсивно або за допомогою стеку.

Пошук у ширину передбачає послідовне відвідування всіх вершин, суміжних із поточною, після чого здійснюється перехід до наступного рівня графа. Для реалізації цього підходу використовується черга. Пошук у ширину дає змогу визначати найкоротші шляхи у незважених графах за кількістю ребер.

Ефективність обходу графа залежить від способу його подання. Для списку суміжності алгоритми обходу мають часову складність $O(V + E)$, де V – кількість вершин, а E – кількість ребер. Для матриці суміжності витрати можуть бути більшими при обробці розріджених графів.

Таким чином, вибір способу подання графа та алгоритму його обходу залежить від структури задачі, щільності графа та вимог до ефективності реалізації.

Практичне завдання

1. Ознайомитися з поняттям графа як структури даних.
2. Розглянути відмінності між орієнтованими та неорієнтованими графами.

3. Реалізувати подання графа за допомогою матриці суміжності.
4. Реалізувати подання графа за допомогою списку суміжності.
5. Створити приклад графа із заданою кількістю вершин і ребер.
6. Реалізувати алгоритм обходу графа в глибину.
7. Реалізувати алгоритм обходу графа в ширину.
8. Перевірити правильність роботи алгоритмів обходу.
9. Визначити порядок відвідування вершин при різних способах обходу.
10. Порівняти особливості реалізації обходу для різних способів подання графа.
11. Оцінити часову складність алгоритмів обходу.
12. Зробити висновки щодо доцільності використання різних способів подання графа.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно засвоїти основні поняття теорії графів, зокрема вершину, ребро, суміжність та шлях. Слід також розуміти, що граф може бути поданий різними способами, кожен із яких має власні переваги та обмеження.

На першому етапі доцільно реалізувати граф у вигляді матриці суміжності. Для цього необхідно створити двовимірний масив, у якому значення елементів відображатимуть наявність зв’язку між вершинами. Після цього слід реалізувати той самий граф у вигляді списку суміжності, де для кожної вершини зберігатиметься перелік її сусідів.

Далі необхідно виконати побудову прикладу графа та переконатися у правильності його подання в обох формах. Особливу увагу слід приділити тому, щоб для неорієнтованого графа зв’язки відображалися симетрично.

Після цього потрібно реалізувати пошук у глибину. Для цього можна використати рекурсивний підхід або стек. Необхідно передбачити структуру для позначення вже відвіданих вершин, щоб уникнути повторного відвідування та зациклення.

Наступним етапом слід реалізувати пошук у ширину. Для цього доцільно використати чергу та окрему структуру для фіксації відвіданих вершин. У процесі обходу потрібно зафіксувати порядок відвідування вершин і порівняти його з результатами пошуку у глибину.

Після реалізації алгоритмів необхідно виконати аналіз ефективності для обох способів подання графа. Слід визначити, у яких випадках матриця суміжності є зручнішою, а коли більш доцільним є список суміжності.

На завершальному етапі необхідно сформулювати висновки щодо особливостей подання графів та реалізації їх обходу у програмних системах.

Контрольні питання

1. Що таке граф як структура даних?
2. Чим орієнтований граф відрізняється від неорієнтованого?
3. Що таке матриця суміжності?
4. Що таке список суміжності?
5. Які переваги має матриця суміжності?
6. Які переваги має список суміжності?
7. У чому полягає сутність пошуку у глибину?
8. У чому полягає сутність пошуку у ширину?

9. Які структури даних використовуються для реалізації DFS і BFS?
10. Яку часову складність мають алгоритми обходу графа?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– теоретичні відомості про способи подання графа;
– реалізація графа у вигляді матриці суміжності;
– реалізація графа у вигляді списку суміжності;
– реалізація обходу в глибину;
– реалізація обходу в ширину;
– аналіз результатів;
– висновки.

Практична робота 14. Реалізація алгоритмів на графах

Мета роботи – ознайомитися з базовими алгоритмами розв'язання задач на графах, навчитися реалізовувати алгоритми пошуку найкоротших шляхів і побудови мінімального остовного дерева та виконувати аналіз їх ефективності.

Ключові положення

Графи є універсальним засобом подання складних структур зв'язків між об'єктами. Якщо базові алгоритми обходу графа дозволяють досліджувати його структуру, то алгоритми на графах дають змогу розв'язувати прикладні задачі, пов'язані з пошуком маршрутів, оптимізацією зв'язків, визначенням вартості проходження та побудовою ефективних схем з'єднання. Такі задачі часто виникають у транспортних системах, комп'ютерних мережах, логістиці, системах маршрутизації та програмній інженерії.

Однією з найпоширеніших задач є пошук найкоротшого шляху між вершинами графа. Для незважених графів найкоротший шлях за кількістю ребер може бути знайдений за допомогою пошуку в ширину. Для зважених графів, у яких кожне ребро має числову вагу, використовуються спеціальні алгоритми, зокрема алгоритм Дейкстри. Його принцип полягає у послідовному виборі вершини з найменшою поточною відстанню від початкової вершини та уточненні відстаней до суміжних вершин.

Алгоритм Дейкстри є ефективним для графів із невід'ємними вагами ребер. У процесі його роботи формується множина вершин, для яких вже визначено найкоротшу відстань, та поступово уточнюються оцінки для інших вершин. Цей алгоритм широко використовується у задачах навігації та маршрутизації.

Іншою важливою задачею є побудова мінімального остовного дерева. Остовне дерево зв'язного неорієнтованого графа є підграфом, що містить усі вершини графа, не має циклів і забезпечує зв'язність. Мінімальне остовне дерево має найменшу можливу сумарну вагу ребер серед усіх можливих остовних дерев.

Для побудови мінімального остовного дерева використовуються жадібні алгоритми, зокрема алгоритми Крускала та Прима. Алгоритм Крускала передбачає впорядкування ребер за зростанням ваги та послідовне додавання до результату найменших ребер, які не

утворюють цикл. Алгоритм Прима починає побудову з однієї вершини та на кожному кроці додає ребро найменшої ваги, що з’єднує вже побудовану частину дерева з новою вершиною.

Ефективність алгоритмів на графах залежить від способу подання графа, кількості вершин і ребер, а також від використаних допоміжних структур даних. Для задач пошуку найкоротших шляхів і побудови мінімального остовного дерева важливе значення мають структури для зберігання множини відвіданих вершин, поточних відстаней та списків ребер.

Таким чином, алгоритми на графах є важливою складовою підготовки фахівця з програмної інженерії, оскільки дозволяють розв’язувати широкий клас задач, пов’язаних з аналізом зв’язків, оптимізацією маршрутів і побудовою ефективних структур.

Практичне завдання

1. Ознайомитися з основними задачами, що розв’язуються на графах.
2. Підготувати зважений граф для виконання алгоритмів.
3. Реалізувати подання графа у зручній для обробки формі.
4. Реалізувати алгоритм пошуку найкоротшого шляху у незваженому графі.
5. Реалізувати алгоритм Дейкстри для зваженого графа.
6. Визначити найкоротші відстані від початкової вершини до інших вершин.
7. Реалізувати алгоритм Крускала або алгоритм Прима для побудови мінімального остовного дерева.
8. Визначити ребра, що входять до мінімального остовного дерева.
9. Обчислити сумарну вагу побудованого остовного дерева.
10. Перевірити коректність роботи реалізованих алгоритмів на тестових прикладах.
11. Оцінити часову складність реалізованих алгоритмів.
12. Зробити висновки щодо особливостей їх застосування.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно повторити основні поняття теорії графів, зокрема вершину, ребро, вагу ребра, шлях, цикл та зв’язність. Особливу увагу слід приділити відмінності між задачами пошуку шляху та задачами побудови остовного дерева, оскільки вони мають різне призначення.

На першому етапі необхідно підготувати приклад зваженого графа та обрати спосіб його подання. Для більшості алгоритмів доцільно використати список суміжності або список ребер. Якщо потрібно працювати з вагами ребер, слід передбачити збереження не лише суміжних вершин, а й числових значень ваг.

Далі доцільно реалізувати алгоритм пошуку найкоротшого шляху. Для незваженого графа можна використати пошук у ширину, а для зваженого – алгоритм Дейкстри. Під час реалізації алгоритму Дейкстри необхідно зберігати поточні оцінки відстаней до вершин та щоразу обирати вершину з найменшим значенням серед ще не оброблених.

Після цього слід реалізувати алгоритм побудови мінімального остовного дерева. Якщо використовується алгоритм Крускала, необхідно впорядкувати всі ребра за вагою та забезпечити перевірку, чи не призведе додавання чергового ребра до утворення циклу. Якщо використовується алгоритм Прима, потрібно організувати поступове розширення множини вершин дерева шляхом вибору ребра найменшої ваги.

У процесі тестування необхідно перевірити правильність обчислення найкоротших шляхів та сумарної ваги мінімального остовного дерева. Доцільно використати невеликі графи, для яких результат можна перевірити вручну, а потім перейти до складніших прикладів.

Особливу увагу потрібно приділити аналізу ефективності алгоритмів. Слід визначити, як кількість вершин і ребер впливає на час виконання. Необхідно також врахувати, що ефективність реалізації залежить від використаних допоміжних структур даних.

На завершальному етапі потрібно зробити висновки щодо доцільності використання конкретних алгоритмів для різних типів задач на графах, зокрема для задач маршрутизації, мережевої оптимізації та аналізу структурних зв’язків.

Контрольні питання

1. Які основні задачі розв’язуються за допомогою алгоритмів на графах?
2. У чому полягає задача пошуку найкоротшого шляху?
3. У яких випадках можна використовувати пошук у ширину для знаходження найкоротшого шляху?
4. У чому полягає принцип роботи алгоритму Дейкстри?
5. Яка умова обмежує застосування алгоритму Дейкстри?
6. Що таке остовне дерево графа?
7. Що таке мінімальне остовне дерево?
8. У чому полягає принцип роботи алгоритму Крускала?
9. У чому полягає принцип роботи алгоритму Прима?
10. Які структури даних використовуються під час реалізації алгоритмів на графах?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– теоретичні відомості про алгоритми на графах;
– опис подання графа;
– реалізація алгоритму пошуку найкоротшого шляху;
– реалізація алгоритму побудови мінімального остовного дерева;
– результати тестування;
– аналіз ефективності алгоритмів;
– висновки.

Практична робота 15. Виявлення компонент зв’язності та топологічне впорядкування графа

Мета роботи – ознайомитися з алгоритмами визначення компонент зв’язності та топологічного впорядкування, навчитися реалізовувати їх засобами мови програмування та аналізувати їх ефективність.

Ключові положення

У задачах аналізу графів важливим є визначення структури зв’язків між вершинами. Однією з базових характеристик неорієнтованого графа є компоненти зв’язності. Компонента зв’язності є підмножиною вершин графа, у якій будь-які дві вершини з’єднані шляхом. Якщо граф містить декілька таких підмножин, він є незв’язним.

Визначення компонент зв’язності дозволяє аналізувати структуру графа, виділяти ізольовані частини та оцінювати ступінь зв’язаності системи. Для знаходження компонент зв’язності використовуються алгоритми обходу графа, зокрема пошук у глибину або пошук у ширину. Кожен запуск алгоритму обходу з нової невідвіданої вершини дозволяє виділити окрему компоненту.

Для орієнтованих графів аналогічним поняттям є сильно зв’язані компоненти. Вони визначаються як множини вершин, між якими існують шляхи в обох напрямках. Для їх

знаходження застосовуються спеціалізовані алгоритми, наприклад алгоритм Косараджу або алгоритм Тар'яна.

Іншою важливою задачею є топологічне впорядкування орієнтованого ациклічного графа. Топологічне впорядкування визначає такий порядок вершин, при якому для кожного ребра початкова вершина передує кінцевій. Ця задача виникає у плануванні виконання задач, аналізі залежностей між модулями програмного забезпечення та організації обчислювальних процесів.

Топологічне впорядкування може бути реалізоване за допомогою обходу в глибину або за допомогою алгоритму, що базується на поступовому видаленні вершин із нульовим вхідним степенем. Обидва підходи забезпечують лінійну часову складність відносно кількості вершин і ребер.

Таким чином, аналіз компонент зв'язності та виконання топологічного впорядкування дозволяє глибше досліджувати структуру графів та вирішувати прикладні задачі, пов'язані з аналізом залежностей і структурних зв'язків.

Практичне завдання

1. Ознайомитися з поняттям компонент зв'язності графа.
2. Реалізувати алгоритм пошуку компонент зв'язності для неорієнтованого графа.
3. Побудувати граф із декількома компонентами зв'язності.
4. Визначити кількість компонент зв'язності та склад кожної з них.
5. Ознайомитися з поняттям сильно зв'язаних компонент для орієнтованого графа.
6. Реалізувати алгоритм знаходження сильно зв'язаних компонент.
7. Побудувати орієнтований граф для тестування алгоритму.
8. Реалізувати алгоритм топологічного впорядкування.
9. Перевірити коректність топологічного порядку для ациклічного графа.
10. Проаналізувати випадки, коли топологічне впорядкування неможливе.
11. Оцінити часову складність реалізованих алгоритмів.
12. Зробити висновки щодо їх застосування.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно повторити поняття зв'язності графа та відмінності між неорієнтованими й орієнтованими графами. Особливу увагу слід приділити тому, що у неорієнтованому графі зв'язність визначається наявністю шляху між вершинами, а в орієнтованому – напрямком ребер.

На першому етапі необхідно реалізувати алгоритм пошуку компонент зв'язності. Для цього доцільно використати пошук у глибину або пошук у ширину, виконуючи обхід графа з кожної невідвіданої вершини. Результатом кожного обходу є окрема компонента.

Далі слід побудувати тестовий граф, який містить декілька ізольованих підграфів. Це дозволить перевірити коректність реалізації алгоритму.

Після цього необхідно перейти до орієнтованих графів та реалізувати алгоритм знаходження сильно зв'язаних компонент. Для цього потрібно виконати декілька обходів графа з урахуванням напрямків ребер.

Наступним етапом є реалізація топологічного впорядкування. Необхідно забезпечити, щоб граф був ациклічним, оскільки наявність циклів робить топологічне впорядкування неможливим.

У процесі тестування слід перевірити правильність визначення компонент та коректність отриманого топологічного порядку. Особливу увагу потрібно приділити випадкам із циклами.

На завершальному етапі необхідно виконати аналіз ефективності алгоритмів та сформулювати висновки щодо їх застосування у задачах програмної інженерії.

Контрольні питання

1. Що таке компонента зв’язності?
2. Як визначити компоненти зв’язності у графі?
3. Що таке сильно зв’язані компоненти?
4. Які алгоритми використовуються для їх знаходження?
5. Що таке топологічне впорядкування?
6. Для яких графів воно можливе?
7. У чому полягає принцип топологічного сортування?
8. Яка складність алгоритмів пошуку компонент зв’язності?
9. Чому топологічне впорядкування неможливе при наявності циклів?
10. У яких задачах застосовуються ці алгоритми?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– теоретичні відомості про компоненти зв’язності та топологічне впорядкування;
– реалізація алгоритмів;
– результати тестування;
– аналіз ефективності;
– висновки.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

Основна

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

Допоміжна

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

Інформаційні ресурси

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

Навчальне видання

Мірошник Марина Анатоліївна

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Методичні вказівки до практичних робіт для здобувачів вищої освіти,
які навчаються за спеціальностями галузі «Інформаційні технології»

Українською мовою