УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Кафедра інформаційних технологій та програмної інженерії

КУРСОВА РОБОТА

з Об’єктно-орієнтованого програмування

на тему: Клас для обслуговування структур типу «Черга з пріоритетами».

Студента II курсу____групи

спеціальності 121 «Інженерія програмного забезпечення»

Стрілецький В.Т.
(прізвище та ініціали)

Керівник _____к.т.н., Пашкевич О.П.

Національна шкала _____

Кількість балів _____ Оцінка: ECTS _____

м. Івано-Франківськ

2021

Університет Короля Данила
назва вищого навчального закладу

Кафедра інформаційних технологій та програмної інженерії

Дисципліна Об’єктоно-орієнтоване програмування

Спеціальність 121 «Інженерія програмного забезпечення»

Курс II Група ІПЗс-2019 Семестр 4

ЗАВДАННЯ

на курсовий проект (роботу) студента

Стрілецький Віталій Тарасович

Прізвище, ім’я, по-батькові

1. Тема проекту (роботи) Клас для обслуговування структур типу «Черга з пріоритетами»

2. Строк здачі студентом закінченого проекту (роботи) _____

3. Вихідні дані до проекту (роботи)

4. Зміст роботи (перелік питань по розділах, які потрібно розробити)

_____

_____

5. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів курсового проекту (роботи)	Строк виконання	Примітки
1	Ознайомився з завданням	16-17.05	
2	Створив основний клас	17-18.05	
3	Ознайомився з методами	19.05	
4	Зробив першу половину	20.05	
5	Зробив другу половину	21.05	
6	Оформлення курсової роботи	22.05	
7	Оформлення курсової роботи	23.05	

Студент _____ _____
Підпис Прізвище, ініціали

Керівник _____ _____
Підпис Прізвище, ініціали

«____» _____ 202___ р.

ЗМІСТ

ВСТУП

В даній курсовій роботі ми описуємо одну з найбільш актуальних на сьогоднішній день мов програмування - мова Java. Вона легка у вивченні, дозволяє створювати програми, які можуть виконуватися на будь-якій платформі без яких-небудь доопрацювань (кросплатформеність). орієнтована на Internet, і найпоширеніше її застосування - невеликі програми, аплети, які запускаються в браузері і є частиною HTML-сторінок. Також мова Java є об'єктно-орієнтованою і поставляється з досить об'ємною бібліотекою класів. На даний момент мова програмування Java є однією з самих кращих мов програмування, якою користуються серйозним програмістам.

Java – це розвинена платформа, що дозволяє використовувати всі сучасні надбання в галузі програмування, а саме: − повністю об'єктно–орієнтована парадигма програмування; − використання віртуальної машини для зменшення часу збірки програмного коду; − незалежність розробленої програми від операційної системи; − підтримка безлічі потоків на концептуальному рівні

(ефективно для багатоядерних процесорів); − відсутність потенційно небезпечних механізмів прямої роботи з пам'яттю, які можуть призвести до переповнення буфера, неконтрольованого виходу за межі масиву і т.п.; − багатий набір стандартних бібліотек для побудови графічного інтерфейсу, обробки тексту за допомогою мови регулярних виразів, 4 математичних обчислень високої точності, передачі даних з мережевих протоколів, шифрування і дешифрування інформації, роботи з реляційними базами даних, роботи з даними в XML форматі і т.п.; − наявність значної кількості безкоштовних бібліотек сторонніх розробників, в тому числі необхідних в автоматизації (бібліотека роботи за протоколом OPC – UTGART, протоколом ModBUS – jamod, бібліотека роботи з різними протоколами, прийнятими в автоматиці, і обладнанням – Java for Process Control.

1.ОСНОВНІ ВИМОГИ ДЛЯ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА JAVA

1.1 Основні етапи розробки Java-програм

Використання грамотної методики при розробці програм необхідне для скорочення часу на усунення помилок і додавання нової функціональності. Тому важливо дотримуватися основних етапів розробки програм.

1. Визначення завдань. На самому початку необхідно визначити перелік завдань і функцій, які мають виконувати програму. Даний список поділяють мінімум на дві частини: вимагає обов'язкового виконання і може бути виконано 6 в майбутньому. Основними завданнями проекту є: завантаження інформації про структуру об'єкта і регулятора, моделювання роботи лінійно–квадратичного регулятора в замкненій САР, відображення стану ТП з можливістю оператора впливати на його хід,ведення графіка перехідних процесів.

2. Складання послідовності виконання дій. На даному етапі розробляються основні алгоритми виконання завдань і послідовності їх виконання.

3. Визначення необхідних ресурсів. Для виконання низки завдань необхідні додаткові ресурси: алгоритми, файли, програмні засоби і т.п. Необхідними ресурсами для роботи програми є розраховані у середовищі Mathwork Matlab матриці лінійно-квадратичного регулятора і об'єкта управління, а також алгоритми роботи лінійно-квадратичного регулятора в замкненій САР.

4. Розробка макета користувача інтерфейсу. Правильно розроблений інтерфейс користувача є важливою частиною програми. Програмний інтерфейс складається з чотирьох вікон: вікно для відображення стану ТП, вікно з інформацією про ТП і розробника, вікно для встановлення завдання на керовані параметри ТП, вікно для відображення графіків.

5. Вибір технологій. Технологія Java у свою чергу складається з величезного набору різних технологій. Виконання кожного завдання може бути виконано за допомогою однієї або декількох технологій. Так, для програмування графічних програм в Java як правило використовуються дві стандартні бібліотеки графічних компонентів: AWT і Swing та альтернативна SWT від IBM. Для розробки даної програми необхідно використовувати стандартні бібліотеки Java – awt, swing, text, net, io, imageio і додаткові – jama, swixml, jfreechart.

6. Вибір середовища розробки. Існує значна кількість безкоштовних і платних інструментів розробки, що відрізняються можливостями і швидкістю роботи. Достатнім для реалізації програми є середовище розробки BlueJ. Її переваги – швидкість, невимогливість до ресурсів, наявність інструментів для налагодження та зручний інтерфейс користувача.

7. Створення програми. Спираючись на реалізовані попередні етапи можливо розпочати до безпосереднього програмування, не відволікаючись на сторонні завдання.

8. Тестування. Для забезпечення надійності та коректності роботи необхідно тестування програми за різних умов та ввідних даних. Програма повинна забезпечувати стабільність роботи за будь–яких можливих умов роботи.

9. Визначення способів поширення програми. На даному етапі визначається, яким чином програма буде надаватися кінцевому користувачеві і розгортатися на клієнтському комп'ютері. Для поширення даного додатка досить використовувати технологію JAR. JAR – це архівний формат, який використовується для агрегації безлічі класів Java програми та пов'язаних з ним даних в один файл з розширенням Jar із зазначенням класу, який містить метод main, з якого починається виконання програми. У середовищі BlueJ JAR файл 7 легко створюється з проекту за допомогою виконання команди меню Project– Create JAR File.

1.2 Особливості використання базових концепцій ООП в Java

Наведені нижче терміни та угоди є загальноприйнятими при програмуванні мовою Java.

1 Пакет (Package) – це сукупність класів і підпакету, об'єднаних спільним ім'ям. Імена пакетів прийнято записувати в нижньому регістрі, наприклад java.util.

2 Клас (Class) – це базова сутність ООП, що володіє певними властивостями. Будь-яка програма мовою Java являє собою клас. Імена класів прийнято іменувати з великої літери і записувати змішаним регістром (кожне нове слово пишеться разом і також з великої літери). Приклади запису – String, StringTokenizer і т.п.

3 Поле (Field / Attribute) – це іменована властивість класу або об'єкта. Поле може ставитися як до кожного об’єкта, так і до класу в цілому. Імена змінних і полів починаються з маленької літери і записуються змішаним регістром, наприклад name, calculateSquare.

4 Об'єкт (Object) – це змінна, типом якої є відповідний клас. Об'єкт також називають екземпляром класу.

5 Метод (Method) – це програмна функція, що відноситься до певного об’єкта або класу. Області (scope), звідки метод може бути доступний, визначаються модифікаторами методу.

Імена методів починаються з маленької літери і записуються змішаним регістром (mixed case). Як правило в іменах методів використовують префікси set

(метод встановлює значення), get (метод повертає значення) і is (метод перевіряє наявність). Наприклад, setName, getName, checkField, isEmpty.

6. Клас може запозичувати методи іншого класу. У мові Java наступництво (inheritance) проводиться за допомогою ключового слова extends public class Square extends Figure {...} 8

7. Модифікатори доступу є реалізацією принципу інкапсуляції в мові Java. Змінюючи модифікатори, можна контролювати область видимості полів, методів, класів.

Основні модифікатори полів:

− private – поле не може бути використано ніде крім даного класу або його екземпляра;

− protected – поле не може бути використано ніде крім даного пакету і в настіпників;

− public – поле доступно звідусіль; − відсутній – поле доступне тільки з поточного пакета;

− static – поле належить структурі класу, одне значення притаманне всім екземплярам;

− final – поле не може бути змінено.

Основні модифікатори методів:

− private – метод не може бути використаний нізвідки крім даного класу (його об'єкта);

− protected – метод не може бути використаний нізвідки крім даного класу (його об'єкта), поточного пакету і всіх його настіпників (їх об'єктів);

− public – метод доступний з будь–якого пакета (публічний API); − відсутній – метод доступний тільки з даного пакету;

− final – метод не може бути перевизначений в настіпника;

− static – метод належить класу;

− abstract – метод не має реалізації.

Основні модифікатори класів:

− відсутній - клас доступний тільки в поточному пакеті;

− public - клас доступний з будь–якого пакета (публічний API);

− final - клас не може мати наступників;

− abstract - клас є абстрактним, не можна створити об'єкт цього класу;

− static - припустимо тільки для вкладених класів. Внутрішній клас є статичним членом зовнішнього класу.

8. Конструктор (constructor) – блок інструкцій, що створює екземпляр класу. Не має заданого значення, що повертається. Має те саме ім'я, що й клас. Зверніть увагу, що в класі завжди присутній конструктор за замовчуванням, якщо явно конструктор не заданий.

9. Виклик методу (method call) – це звернення до члена класу за його ім’ям. Результат виклику методу – виконані оператори і повертається значення (якщо вказано).

10. Клас, описаний всередині іншого класу називається внутрішнім (inner). Може бути використаний для зручності, обмеження області його видимості і для приховування реалізації

1.3 Структура Java програми

Java–програма складається з одного або декількох визначень класів, розміщених в одному або декількох файлах (суфікс імені файла –. Java). Для компіляції програм використовується java–компілятор javac, наприклад, javac TestClass.java

У результаті для кожного класу з вихідного з вихідного файла створюється файл класу, що містить байт–коди класу. Основа імені файла класу збігається з іменем класу, до неї додається суфікс Class.

Один із класів програми повинен мати модифікатор public і містити метод main, з якого починається виконання програми.

Для виконання програми необхідно запустити інтерпретатор java, вказавши йому ім'я класу, що містить метод main, наприклад, java TestClass

Метод main в Java має наступний прототип: public static void main (String argv []), де argv – це масив рядків, що являють собою аргументи командного рядка виклику інтерпретатора і розташовуються в ньому після імені класу.

2.ОСНОВНІ ПРИНЦИПИ ООП

2.1. Інкапсуляція.

Одним з визначальних факторів при проектуванні компонентів програми є приховування внутрішніх даних компоненту і деталей його реалізації від інших компонентів програми та надання набору методів для взаємодії з ним (API). Цей принцип є одним з чотирьох фундаментальних принципів ООП і називається інкапсуляцією.

Правильна інкапсуляція має велике значення з багатьох причин**:**

1. Вона сприяє повторному використанню компонентів: оскільки в цьому випадку компоненти взаємодіють між собою лише через їх API і нечутливі до змін внутрішньої структури, вони можуть використовуватись в більш широкому контексті.

2. Інкапсуляція пришвидшує процес розробки: слабко пов'язані один з одним компоненти (тобто компоненти, чий код якомога менше звертається або використовує код інших компонентів) можуть розроблятися, тестуватися та доповнюватися незалежно.

3. Правильно інкапсульовані компоненти більш зрозумілі та легше налагоджуються, що спрощує підтримку програми.

У мові Java інкапсуляція реалізована за допомогою системи класів, які дозволяють зібрати інформацію про об'єкт в одному місці; пакетів, які групують класи по певному критерію, і модифікаторів доступу, якими можна позначити весь клас або його поле чи метод.

Всього існує чотири модифікатори доступу:
- public – повний доступ до сутності (полю або методу класу) з будь-якого пакету;
- protected – доступ до сутності лише для класів свого пакету і нащадків класу;
- private – доступ тільки всередині класу, в якому оголошена сутність;
- неявний модифікатор за замовчуванням (за відсутності трьох явних) – доступ до сутності лише для класів свого пакету.

Для досягнення правильної інкапсуляції також необхідно надати коректний API для роботи з компонентом. Наприклад, в сеттер для змінної можна включити логіку перевірки значень, які передаються, або не надавати сеттери в класі взагалі, якщо клас повинен бути доступним лише для читання.

2.2. Наслідування

Наслідування є одним з найвагоміших принципів об'єктно-орієнтованого програмування, оскільки воно дозволяє створювати ієрархічні структури об'єктів. Використовуючи наслідування можна створити загальний клас, який буде визначати характеристики і поведінку, властиві певному набору пов'язаних об'єктів. В подальшому цей клас може наслідуватися іншими, другорядними класами, кожен з яких додаватиме унікальні, властиві лише йому характеристики і доповнюватиме або змінюватиме поведінку базового класу. В термінах Java такий загальний клас називається суперкласом (superclass) або базовим класом (base class), або батьківським класом (parent class), а клас, який його наслідує - підкласом (subclass) або дочірнім класом (child class), або похідним класом. (derived class).

Наслідування реалізує відношення «є» (“is-a”) між суперкласом і підлкасом. Нехай, наприклад, класи Employee та Manager являють собою абстракцію понять «Співробітник» і «Менеджер». Кожний менеджер є також співробітником компанії, в якій він працює, отже клас Manager знаходиться у відношенні “is-a” з класом Employee. Таким чином, з точки зору наслідування, при побудові ієрархії класів, клас Employee буде суперкласом, а клас Manager – дочірнім класом. При цьому клас, який є нащадком якого-небудь класу, може бути суперкласом для одного чи декількох інших класів. Також на відміну від, наприклад, C++, в Java відсутнє множинне наслідування , тобто будь-який клас може мати не більше одного батьківського класу. А всі класи, суперклас котрих явно не вказаний, наслідують клас Object.

Клас Employee у вищезгаданому прикладі, є суперкласом не тому, що він головніший за клас Manager або містить більше функціональності. Насправді, вірно зворотне твердження: функціональність підкласів не вужча, а часто суттєво ширша за функціональність їх батьківських класів. Префікси «супер-» і «під-» прийшли в Java з математики: множина всіх менеджерів міститься в множині всіх співробітників, і, таким чином, є підмножиною множини співробітників.

2.3. Поліморфізм

Розглядаючи поліморфізм необхідно пам'ятати, що цей принцип нерозривно пов'язаний з іншим принципом ООП – наслідуванням, яке допомагає реалізувати поліморфізм. Візьмемо для прикладу абстрактний клас «Автомобіль», який наслідують два конкретних класи – «Спортивний автомобіль» та «Вантажний автомобіль».

І спортивні, і вантажні автомобілі володітимуть спільними характеристиками і матимуть можливість виконувати загальні для всіх автомобілів дії, вказані в абстрактному батьківському класі, але конкретна реалізація цих дій може бути різною.

Наприклад, загальна для всіх автомобілів дія «завестись» у спортивному автомобілі може бути реалізована шляхом натискання кнопки, а у вантажного - за допомогою ключа. Один результат – різні рішення. В цьому і полягає поліморфізм.

Більш точно, поліморфізм - один з принципів ООП, який дозволяє викликом перевизначеного методу через змінну батьківського класу отримати поведінку, яка буде відповідати реальному похідному класу, на який посилається ця змінна.

2.3. Абстракція.

Відносно недавно абстракцію почали виділяти, як самостійний четвертий принцип.

Одне з визначень слова «абстракція», які можна зустріти в сучасних словниках:

Абстракція (від лат. abstractio — виокремлення, відсторонення або відділення) — теоретичний прийом дослідження, який дозволяє відсторонитися від деяких несуттєвих, у певному сенсі, властивостей досліджуваних явищ і виокремити суттєві та визначальні властивості.

Всі мови програмування пропонують їх користувачу деякі абстрації. Так мови групи асемблер є свого роду абстракцією відповідних мікропроцесорів, оскільки дозволяють відволіктися від деталей їх реалізації та звертатися до них через набір більш високорівневих інструкцій. Імперативні мови програмування, які з'явилися після асемблеру, наприклад Basic, Fortran, C, є більш високим рівнем абстракції над асемблерними мовами – вони дають можливість використовувати більш звичні для людини синтаксичні конструкції за рахунок наближення синтаксису до природніх мов. Об'єктно-орієнтовані мови, такі як Java, виводят розробку на ще вищий рівень абстракції: об'єкти в ООП являють собою моделі понять оточуючого світу, такі як Працівник, Сервер, Запис в щоденнику, і виділяють лише властивості цих понять, необхідні в конкретному випадку для вирішення конкретної проблеми. Наприклад, клас Student в програмі обліку студентів університету, крім загальних полів, таких як ім'я, прізвище, дата народження і т.д., міститиме поля, які відображають інформацію про номер залікової книжки, статус студента

(дійсний, академічна відпустка, відраховано), факультет, номер його групи, оцінки за семестр і т. ін. Але для такого ж класа Student в програмі обліку студентів у тренінг-центрі EPAM така інформація буде неактуальна: клас міститиме поля, які відображають навчальний проект, на який був розподілений студент, рівень його англійської мови за результатами останнього тестування, кількість відвіданих заходів та ін.

3. ПОБУДОВА І АНАЛІЗ АЛГОРИТМІВ

3.1. Формалізація алгоритмів.

Процес створення комп'ютерної програми для вирішення будь-якої практичної задачі складається з декількох етапів:

- формалізація і створення технічного завдання на вихідну задачу;
- розробка алгоритму вирішення задачі;
- написання, тестування, наладка і документування програми;
- отримання розв'язку вихідної задачі шляхом виконання програми. Половина справи зроблена, якщо знати, що поставлена задача має вирішення. В першому наближенні більшість задач, які зустрічаються на практиці, не мають чіткого й однозначного опису. Певні задачі взагалі неможливо сформулювати в термінах, які допускають комп'ютерне вирішення. Навіть якщо допустити, що задача може бути вирішена на комп'ютері, часто для її формального опису потрібна велика кількість різноманітних параметрів. І лише в ході додаткових експериментів можна знайти інтервали зміни цих параметрів.

Якщо певні аспекти вирішуваної задачі можна виразити в термінах якоїнебудь формальної моделі, то це, безумовно, необхідно зробити, так як в цьому випадку в рамках цієї моделі можна взнати, чи існують методи й алгоритми вирішення задачі. Навіть якщо такі методи й алгоритми не існують на сьогоднішній день, то застосування засобів і властивостей формальної моделі допоможе в побудові

вирішення вихідної задачі. Практично будь-яку галузь математики або інших наук можна застосувати до побудови моделі певного класу задач.

Для задач, числових за своєю природою, можна побудувати моделі на основі загальних математичних конструкцій, таких як системи лінійних рівнянь, диференціальні рівняння. Для задач з символьними або текстовими даними можна застосувати моделі символьних послідовностей або формальних граматик. Вирішення таких задач містить етапи компіляції і інформаційного пошуку. Коли побудована чи підібрана потрібна модель вихідної задачі, то природно шукати її вирішення в термінах цієї моделі. На цьому етапі основна мета полягає в побудові розв'язку в формі алгоритму, який складається з скінченої послідовності інструкцій, кожна з яких має чіткий зміст і може бути виконана з скінченими обчислювальними затратами за скінчений час. Інструкції можуть виконуватися в алгоритмі будь-яку кількість раз, при цьому вони самі визначають цю кількість повторень. Проте вимагається, щоб при будь-яких вхідних даних алгоритм завершився після виконання скінченої кількості інструкцій. Таким чином, програма, яка написана на основі розробленого алгоритму, при будь-яких початкових даних ніколи не повинна приводити до нескінченних циклічних обчислень.

Є ще один аспект у визначення алгоритмів. Алгоритмічні інструкції повинні мати „чіткий зміст" і виконуватися з „скінченими обчислювальними затратами". Природно, те, що зрозуміло одній людині і має для неї „чіткий зміст", може зовсім інакше представлятися іншій. Те ж саме можна сказати про поняття „скінчених затрат": на практиці часто важко довести, що при будь-яких вихідних даних виконання послідовності інструкцій завершиться, навіть якщо чітко розуміти зміст кожної інструкції. У цій ситуації, враховуючи всі аргументи за і проти, було б корисним спробувати досягнути узгодження про „скінченні затрати" у відношенні до послідовності інструкцій, які складають алгоритм.

3.2. Покрокове проектування алгоритмів.

Оскільки для вирішення вихідної задачі застосовують деяку математичну модель, то тим самим можна формалізувати алгоритм вирішення в термінах цієї моделі. У початкових версіях алгоритму часто застосовуються узагальнені оператори, які потім перевизначаться у вигляді більш дрібних, чітко визначених інструкцій. Але для перетворення неформальних алгоритмів у комп'ютерні програми необхідно пройти через декілька етапів формалізації (цей процес називають покроковою деталізацією), поки не отримають програму, яка повністю складається з формальних операторів мови програмування. Схематично узагальнений процес програмування можна представити наступною схемою. 31 Математична модель Неформальний алгоритм Абстрактний тип даних Програма на псевдомові Структури даних Програма на мові програмування На першому етапі створюється модель вихідної задачі, для чого застосовуються відповідні математичні моделі. На цьому етапі для знаходження рішення також будується неформальний алгоритм. На наступному етапі алгоритм записується на псевдомові – композиції конструкцій мови програмування і менш формальних і узагальнених операторів на простій мові. Продовженням цього етапу є заміна неформальних операторів послідовністю більш детальних і формальних операторів. З цієї точки зору програма на псевдомові повинна бути достатньо детальною, так як в ній фіксуються (визначаються) різні типи даних, над якими виконуються оператори. Потім створюються абстрактні типи даних для кожного зафіксованого типу даних (за винятком елементарних типів даних, таких як цілі й дійсні числа або символьні стрічки) шляхом завдання імен функцій для кожного оператора, який працює з даними абстрактного типа, і заміни їх (операторів) викликом відповідних функцій. Третій етап процесу – програмування – забезпечує реалізацію кожного абстрактного типа даних і створення функцій для виконання різних операторів над даними цих типів. На цьому етапі також замінюються всі неформальні оператори псевдомови на код мови програмування. Результатом цього етапу повинна бути виконувана програма. Після її наладки отримують працюючу програму.

3.3. Характеристики алгоритму.

Охарактеризуємо поняття алгоритму не формально, а дескриптивно за допомогою таблиці: Вирішення гарантоване Так Ні Чи обов’язкове оптимальне вирішення Так Алгоритм Імовірнісний алгоритм Ні Приблизний алгоритм Евристичний алгоритм Як бачимо, алгоритм – це механізм, який не тільки повинен гарантувати те, що вирішення колись буде знайдене, але й те, що буде знайдене саме оптимальне, тобто найкраще вирішення. Крім того, алгоритм повинен мати наступні п’ять якостей:

1. Обмеженість в часі – робота алгоритму обов’язково повинна завершитись через деякий розумний період часу.

2. Правильність – алгоритм повинен знаходити правильне рішення.

3. Детермінованість – скільки б разів не виконувався алгоритм з однаковими вхідними даними, результат повинен бути однаковим.

4. Скінченність – опис алгоритму повинен мати скінчену кількість кроків.

5. Однозначність – кожний крок алгоритму повинен інтерпретуватися однозначно. Як видно з таблиці, евристика – це пряма протилежність класичному алгоритму, так як вона не дає ніяких гарантій на те, що рішення буде знайдене, так само, як і на те, що воно буде оптимальним. Між ними є два перехідні стани – приблизні алгоритми (рішення гарантоване, але його оптимальність – ні) і імовірнісні алгоритми (рішення не гарантоване, але якщо воно буде знайдене, то обов’язково буде оптимальним).

3.4. Складність алгоритму.

У процесі вирішення прикладних задач вибір потрібного алгоритму викликає певні труднощі. І справді, на чому базувати свій вибір, якщо алгоритм повинен задовольняти наступні протиріччя.

1. Бути простим для розуміння, перекладу в програмний код і наладки.

2. Ефективно використовувати комп’ютерні ресурси і виконуватися швидко. Якщо написана програма повинна виконуватися лише декілька разів, то перша вимога найбільш важлива. Вартість робочого часу програміста, звичайно, значно

перевищує вартість машинного часу виконання програми, тому вартість програми оптимізується за вартістю написання (а не виконання) програми. Якщо мати справу з задачею, вирішення якої потребує значних обчислювальних затрат, то вартість виконання програми може перевищити вартість написання програми, особливо якщо програма повинна виконуватися багаторазово. Тому, з економічної точки зору, перевагу буде мати складний комплексний алгоритм (в надії, що результуюча програма буде виконуватися суттєво швидше, ніж більш проста програма). Але і в цій ситуації розумніше спочатку реалізувати простий алгоритм, щоб визначити, як повинна себе вести більш складна програма. При побудові складної програмної системи бажано реалізувати її простий прототип, на якому можна провести необхідні виміри й змоделювати її поведінку в цілому, перш ніж приступати до розробки кінцевого варіанту. Таким чином, програмісти повинні бути обізнані не тільки з методами побудови швидких алгоритмів, але й знати, коли їх потрібно застосувати. Існує декілька способів оцінки складності алгоритмів. Програмісти, звичайно, зосереджують увагу на швидкості алгоритму, але важливі й інші вимоги, наприклад, до розмірів пам’яті, вільного місця на диску або інших ресурсів. Від швидкого алгоритму може бути мало толку, якщо під нього буде потрібно більше пам’яті, ніж встановлено на комп’ютері. Важливо розрізняти практичну складність, яка є точною мірою часу обчислення і об’єму пам’яті для конкретної моделі обчислювальної машини, і теоретичну складність, яка більш незалежна від практичних умов виконання алгоритму і дає порядок величини вартості. Більшість алгоритмів надає вибір між швидкістю виконання і ресурсами. Задача може виконуватися швидше, використовуючи більше пам’яті, або навпаки – повільніше з меншим обсягом пам’яті. Із цього зв’язку випливає ідея просторовочасової складності алгоритмів. При цьому підході складність алгоритму оцінюється в термінах часу і простору, і знаходиться компроміс між ними.

3.5. Ефективність алгоритмів.

Одним із способів визначення часової ефективності алгоритмів полягає в наступному: на основі даного алгоритму потрібно написати програму і виміряти час її виконання на певному комп'ютері для вибраної множини вхідних даних. Хоча такий спосіб популярний і, безумовно, корисний, він породжує певні проблеми. Визначений час виконання програми залежить не тільки від використаного алгоритму, але й від архітектури і набору внутрішніх команд даного комп'ютера, від якості компілятора, і від рівня програміста, який реалізував даний алгоритм. Час виконання також може суттєво залежати від вибраної множини тестових вхідних даних. Ця залежність стає очевидною при реалізації одного й того ж алгоритму з використанням різних комп'ютерів, різних компіляторів, при залучені програмістів різного рівня і при використанні різних тестових даних. Щоб підвищити об'єктивність оцінки алгоритмів прийняли асимптотичну часову складність як основну міру ефективності виконання алгоритму. Говорять, що час виконання алгоритму має порядок T(N) від вхідних даних розміру N. Одиниця вимірювання T(N) не визначена, але під нею розуміють кількість інструкцій, які виконуються на ідеалізованому комп'ютері. Для багатьох програм час виконання дійсно є функцією вхідних даних, а не їх розміру. У цьому випадку визначають T(N) як час виконання в найгіршому випадку, тобто, як максимум часів виконання за всіма вхідними даними розміру N. Поряд з тим розглядають Tср(N) як середній (в статистичному розумінні) час виконання за всіма вхідними даними розміру N. Хоча Tср(N) є достатньо об'єктивною мірою виконання, але часто неможливо передбачити, або обґрунтувати, рівнозначність усіх вхідних даних. На практиці середній час виконання знайти складніше, ніж найгірший час виконання, так як математично це зробити важко і, крім цього, часто не буває простого визначення поняття „середніх" вхідних даних. Тому, в основному, користуються найгіршим часом виконання як міра часової складності алгоритмів. Продуктивність алгоритму оцінюють за порядком величини. Говорять, що алгоритм має складність порядку , якщо час виконання алгоритму росте пропорційно функції із збільшенням розмірності початкових даних N . O – позначає „величина порядку". Приведемо деякі функції, які часто зустрічаються

при оцінці складності алгоритмів. Ефективність степеневих алгоритмів звичайно вважається поганою, лінійних – задовільній, логарифмічних – хорошою. Функція Примітка f(N)=C C – константа f(N)=log(log(N)) f(N)=log(N) f(N)=NC C – константа від нуля до одиниці f(N)=N f(N)=N*log(N) f(N)=NC C – константа більша одиниці f(N)=CN C – константа більша одиниці f(N)=N! тобто 1*2* … N Оцінка з точністю до порядку дає верхню межу складності алгоритму. Те, що програма має певний порядок складності, не означає, що алгоритм буде дійсно виконуватися так довго. При певних вхідних даних, багато алгоритмів виконується набагато швидше, ніж можна припустити на підставі їхнього порядку складності. У числових алгоритмах точність і стійкість алгоритмів не менш важлива, ніж їх часова ефективність.

3.6. Правила аналізу складності алгоритмів.

У загальному випадку час виконання оператора або групи операторів можна розглядати як функцію з параметрами – розміром вхідних даних і/або одної чи декількох змінних. Але для часу виконання програми в цілому допустимим параметром може бути лише розмір вхідних даних. Час виконання операторів присвоєння, читання і запису має порядок O(N^2). Час виконання послідовності операторів визначається за правилом сум. Тому міра росту часу виконання послідовності операторів без визначення констант пропорційності співпадає з найбільшим часом виконання оператора в даній послідовності. Час виконання умовних операторів складається з часу виконання умовно виконуваних операторів і часу обчислення самого логічного виразу. Час обчислення логічного виразу часто має порядок O(N^2) . Час для всієї умовної конструкції складається з часу обчислення логічного виразу і найбільшого з часів, який необхідний для виконання операторів, що виконуються при різних значеннях логічного виразу. Час виконання циклу є сумою часів усіх часів виконуваних конструкцій циклу, які в свою чергу складаються з часів виконання операторів тіла циклу і часу обчислення умови завершення циклу. Часто час виконання циклу обчислюється, нехтуючи визначенням констант пропорційності, як добуток кількості виконуваних операцій циклу на найбільший можливий час виконання тіла циклу.

Час виконання кожного циклу, якщо в програмі їх декілька, повинен визначатися окремо.

4. АЛГОРИТМИ СОРТУВАННЯ

4.1. Задача сортування.

Для самого загального випадку задачу сортування формулюється: є деяка неврегульована вхідна множина ключів і потрібно отримати множину цих же ключів, впорядкованих за збільшенням або зменшенням. Зі всіх задач програмування сортування, можливо, має найбагатший вибір алгоритмів розв'язку. Назвемо деякі чинники, які впливають на вибір алгоритму.

1. Наявний ресурс пам'яті: повинні вхідна й вихід множини розташовуватися в різних ділянках пам'яті, чи вихідна множина може бути сформована на місці вхідної. В останньому випадку наявна ділянка пам'яті повинна в ході сортування динамічно перерозподілятися між вхідною і вихідною множинами.

2. Початкова впорядкованість вхідної множини: у вхідній множині можуть попадатися впорядковані ділянки. В граничному випадку вхідна множина може виявитися вже впорядкованою. Одні алгоритми не враховують початкової впорядкованості і вимагають одного і того ж часу для сортування будь-якої множини даного обсягу, інші виконуються тим швидше, чим краще впорядкованість на вході.

3. Часові характеристики операцій: при визначенні порядку алгоритму час виконання вважається звичайно пропорційним кількості порівнянь ключів. Ясно, проте, що порівняння числових ключів виконується швидше, ніж стрічкових, операції пересилки, характерні для деяких алгоритмів, 35 виконуються тим швидше, ніж менший об'єм записів. Залежно від характеристик запису таблиці може бути вибраний алгоритм, що забезпечує мінімізацію числа тих чи інших операцій.

4. Складність алгоритму. Простий алгоритм вимагає меншого часу для його реалізації і вірогідність помилки в реалізації його менше. При програмуванні вимоги дотримання термінів розробки і надійності продукту можуть навіть превалювати над вимогами ефективності функціонування. Алгоритм сортування називається усталеним, якщо у відсортованому масиві він не змінює порядку розташування елементів.

Ефективність методів сортування визначається двома параметрами:

- кількістю порівнянь;
- кількістю пересилань елементів.

Різноманітність алгоритмів сортування вимагає деякої їхньої класифікації. Вибраний один з вживаних для класифікації підходів, орієнтований перш за все на логічні характеристики використовуваних алгоритмів. Згідно цьому підходу будьякий алгоритм сортування використовує одну з наступних чотирьох стратегій (або їхню комбінацію).

1. Стратегія вибірки. З вхідної множини вибирається наступний за критерієм впорядкованості елемент і включається в вихідну множину на наступне з місце.

2. Стратегія включення. З вхідної множини вибирається наступний за номером елемент і включається в вихідну множину на те місце, яке він повинен займати відповідно до критерію.

3. Стратегія розподілу. Вхідна множина розбивається на ряд підмножин і сортування ведеться у середині кожної такої підмножини.

4. Стратегія злиття. Вихідна множина отримується шляхом злиття маленьких впорядкованих підмножин.

4.2. Сортування вибіркою.

Даний метод реалізує практично „дослівно” стратегію вибірки. При програмній реалізації алгоритму виникає проблема значення ключа „порожньо”. Досить часто програмісти використовують в якості такого деяке явно відсутнє у вхідній послідовності значення ключа. Інший підхід – створення окремого вектора, кожний елемент якого має логічний тип і відображає стан відповідного елемента вхідної множини.

Алгоритм сортування простою вибіркою рідко застосовується. Набагато частіше застосовується його обмінний варіант. При обмінному сортуванні вибіркою вхідна і вихід множини розташовуються в одній і тій же ділянці пам’яті; вихідна – на початку ділянки, вхідна – в тій частині, що залишилася. У початковому стані вхідна множина займає всю ділянку, а вихідна множина – порожня. У міру виконання сортування вхідна множина звужується, а вихідна – розширяється.

Принцип методу полягає в наступному. Знаходять і вибирають в масиві елементів елемент з мінімальним значенням на інтервалі від першого до останнього елемента і міняють його місцями з першим елементом. На другому кроці знаходять 36 елемент з мінімальним значенням на інтервалі від другого до останнього елемента і міняють місцями його з другим елементом. І так далі для всіх елементів.

Очевидно, що обмінний варіант забезпечує економію пам’яті та при його реалізації не виникає проблема „порожнього” значення. Загальна кількість порівнянь зменшується удвічі – N*(N-1)/2, але порядок алгоритму залишається степеневим. Кількість перестановок N-1, але перестановка удвічі більше потребує часу, ніж пересилка в попередньому алгоритмі.

Досить проста модифікація алгоритму обмінного сортування вибіркою передбачає пошук в одному циклі перегляду вхідної множини відразу і мінімуму, і

максимуму, і обмін їх з першим і з останнім елементами множини відповідно. Хоча сумарна кількість порівнянь і пересилок в цій модифікації не зменшується, досягається економія на кількості ітерацій зовнішнього циклу.

Приведені вище алгоритми сортування вибіркою практично нечутливі до початкової впорядкованості. В будь-якому випадку пошук мінімуму вимагає повного перегляду вхідної множини. В обмінному варіанті початкова впорядкованість може дати деяку економію на перестановках для випадків, коли мінімальний елемент знайдений на першому місці у вхідній множині.

Ще один варіант такого сортування – сортування бульбашкою. При перегляді вхідної множини попарно порівнюються сусідні елементи множини. Якщо порядок їхнього проходження не відповідає заданому критерію впорядкованості, то елементи міняються місцями. В результаті одного такого перегляду при сортуванні за збільшенням елементів елемент з найбільшим значенням ключа переміститься („спливе”) на останнє місце в множині. При наступному проході на своє місце „спливе” другий за величиною ключа елемент і т.д. Вихідна множина, таким чином, формується в кінці сортованої послідовності, при кожному наступному проході його об’єм збільшується на 1, а об’єм вхідної множини зменшується на 1.

Порядок сортування бульбашкою – O(N^2). Середнє число порівнянь – N*(N1)/2 і таке ж середня кількість перестановок, що значно гірше, ніж для обмінного сортування простим вибором. Проте, та обставина, що тут завжди порівнюються і переміщаються тільки сусідні елементи, робить сортування бульбашкою зручним для обробки зв’язних списків.

Ще одна перевага сортування бульбашкою полягає в тому, що при незначних модифікаціях її можна зробити чутливою до початкової впорядкованості вхідної множини.

Ще одна модифікація сортування бульбашкою носить назву шейкерсортування. Суть її полягає в тому, що напрями переглядів чергують: за проходом до кінця множини слідує прохід від кінця до початку вхідної множини. При перегляді в прямому напрямку запис з найбільшим ключем ставиться на своє місце в

послідовності, при перегляді у зворотному напрямі – запис з самим меншим. Цей алгоритм досить ефективний для задач відновлення впорядкованості, коли початкова послідовність вже була впорядкована, але піддалася не дуже значним змінам. Впорядкованість в послідовності з одиночною зміною буде гарантовано відновлена усього за два проходи.

Сортування Шелла – ще одна модифікація сортування бульбашкою. Суть її полягає в тому, що тут виконується порівняння ключів, віддалених один від одного на деяку відстань d. Початковий розмір d звичайно вибирається рівним половині загального розміру сортованої послідовності. Виконується сортування бульбашкою з інтервалом порівняння d. Потім величина d зменшується удвічі і знов виконується сортування бульбашкою, далі d зменшується ще удвічі і т.д. Останнє сортування бульбашкою виконується при d=1. Якісний порядок сортування Шелла залишається O(N^2), середнє ж число порівнянь, визначене емпіричним шляхом, – N*log2(N)^2. Прискорення досягається за рахунок того, що виявленні „не на місці” елементи при d>1, швидше „спливають” на свої місця.

4.3. Сортування включенням.

Цей метод – „дослівна” реалізації стратегії включення. Порядок алгоритму сортування простим включенням – O(N^2), якщо враховувати тільки операції порівняння. Але сортування вимагає ще й в середньому 2 N 4 переміщень, що робить її в такому варіанті значне менш ефективною, ніж сортування вибіркою.

Ефективність алгоритму може бути дещо поліпшена при застосуванні не лінійного, а дихотомічного пошуку. Проте, слід мати на увазі, що таке збільшення ефективності може бути досягнуте лише на значній кількості елементів. Так як алгоритм вимагає великої кількості пересилок, при значному обсязі одного запису ефективність може визначатися не кількістю операцій порівняння, а кількістю пересилок.

Реалізація алгоритму обмінного сортування простими вставками відрізняється від базового алгоритму тільки тим, що вхідна і вихідна множина розміщені в одній ділянці пам’яті.

Бульбашкове сортування включенням – це модифікація обмінного варіанту сортування. В цьому методі вхідна і вихід множини знаходяться в одній послідовності, причому вихід – в початковій її частині. В початковому стані можна вважати, що перший елемент послідовності вже належить впорядкованій вихідній множині, інша частина послідовності – невпорядкована. Перший елемент вхідної множини примикає до кінця вихідної множини. На кожному кроці сортування відбувається перерозподіл послідовності: вихідна множина збільшується на один елемент, а вхідна – зменшується. Це відбувається за рахунок того, що перший елемент вхідної множини тепер вважається останнім елементом вихідної. Потім виконується перегляд вихідної множини від кінця до початку з перестановкою сусідніх елементів, які не відповідають критерію впорядкованості. Перегляд припиняється, коли припиняються перестановки. Це приводить до того, що останній елемент вихідної множини „випливає" на своє місце в множині. Оскільки при цьому перестановка приводить до зсуву нового в вихідній множині елемента на одну позицію ліворуч, немає сенсу кожен раз проводити повний обмін між сусідніми елементами – достатньо зсовувати старий елемент праворуч, а новий елемент записати в вихідну множину, коли його місце буде встановлено.

Хоча обмінні алгоритми стратегії включення і дозволяють скоротити число порівнянь за наявності деякої початкової впорядкованості вхідної множини, значна кількість пересилок істотно знижує ефективність цих алгоритмів. Тому алгоритми включення доцільно застосовувати до зв'язних структур даних, коли операція перестановки елементів структури вимагає не пересилки даних в пам'яті, а виконується способом корекції покажчиків.

Турнірний метод сортування отримав свою назву через схожість з кубковою системою проведення спортивних змагань: учасники змагань розбиваються на пари, в яких розігрується перший тур; з переможців першого туру складаються пари для розиграшу другого туру і т.д.

Алгоритм сортування складається з двох етапів. На першому етапі будується дерево: аналогічне схемі розиграшу кубка. Алгоритм сортування впорядкованим бінарним деревом складається з побудови впорядкованого бінарного дерева і

подальшого його обходу. Якщо немає необхідності в побудові всього лінійного впорядкованого списку значень, то немає необхідності і в обході дерева, в цьому випадку застосовується пошук у впорядкованому бінарному дереві. Відзначимо, що порядок алгоритму – O(N*log2(N)), але в конкретних випадках все залежить від впорядкованості початкової послідовності, який впливає на ступінь збалансованості дерева і нарешті – на ефективність пошуку.

Заслуговує на увагу модифікація цього алгоритму запропонована Р.Флойдом. Метод сортування за допомогою прямої вибірки базується на повторних пошуках найменшого ключа серед N елементів, серед тих що залишилися N-1 елементів і так далі. Удосконалити такий метод сортування можна залишаючи після кожного проходу більше інформації, ніж просто ідентифікація єдиного мінімального елемента. Наприклад, виконавши n/2 порівнянь, можна визначити в кожній парі ключів менший. За допомогою n/4 порівнянь – менший із пари вже вибраних менших і так далі. Провівши n-1 порівнянь, можна побудувати дерево вибору і ідентифікувати його корінь як потрібний найменший ключ.

Другий етап сортування – спуск вздовж шляху, відміченого найменшим елементом, і виключення його з дерева шляхом заміни або на пустий елемент (дірку) в самому низу, або на елемент із сусідньої гілки в проміжних вершинах. Елемент, який перемістився в корінь дерева, знову буде найменшим (тепер вже другим) ключем, і його можна виключити. Після n таких кроків дерево стане пустим і процес сортування завершується.

Звичайно, хотілося б позбавитися дірок, якими в кінцевому рахунку буде заповнене все дерево і які породжують багато непотрібних порівнянь. Крім того, потрібно знайти б таке представлення дерева з n елементів, яке потребує лише n одиниць пам’яті.

Р. Флойдом був запропонований деякий „лаконічний” спосіб побудови піраміди „на тому ж місці”, який використовує функцію зсуву елементів початкового вектора.

Сортування частково впорядкованим бінарним деревом також належить до цієї групи сортування. У бінарному дереві, яке будується при цьому для кожного вузла

справедливе наступне твердження: значення ключа, записане у вузлі, менше, ніж ключі його нащадків. Для повністю впорядкованого дерева є вимоги до співвідношення між ключами нащадків. Для даного дерева таких вимог немає, тому таке дерево і називається частково впорядкованим. Крім того, таке дерево повинно бути абсолютно збалансованим. Це означає не тільки те, що довжини шляхів до будь-якого двох листків розрізняються не більш, ніж на 1, але і те, що при додаванні нового елемента в дерево перевага завжди віддається лівій гілці, поки це не порушує збалансованість.

Для сортування цим методом потрібно визначити дві операції: вставка в дерево нового елемента і вибірка з дерева мінімального елемента; причому виконання будь-якій з цих операцій не повинне порушувати ні сформульованої вище часткової впорядкованості дерева, ні його збалансованості.

Якщо застосовувати сортування частково впорядкованим деревом для впорядкування вже готової послідовності розміром N, то необхідно N раз виконати вставку, а потім N раз – вибірку. Порядок алгоритму – O(N*log2(N)), але середнє значення кількості порівнянь приблизно в 3 рази більше, ніж для турнірного сортування. Але сортування частково впорядкованим деревом має одну істотну перевагу перед всіма іншими алгоритмами – це найзручніший алгоритм для „сортування on-line”, коли сортована послідовність не зафіксована до початку сортування, а міняється в процесі роботи і вставки чергують з вибірками. Кожна зміна (додавання елемента) сортованої послідовності вимагає тут не більш, ніж 2*log2(N) порівнянь і перестановок, в той час, як інші алгоритми вимагають при одиничній зміні нового впорядковування всієї послідовності „за повною програмою”.

4.4. Сортування розподілом.

Алгоритм порозрядного сортування вимагає представлення ключів сортованої послідовності у вигляді чисел в деякій системі числення P. Число проходів сортування рівно максимальному числу значущих цифр в числі – D. При кожному проході аналізується значуща цифра в черговому розряді ключа, починаючи з

молодшого розряду. Всі ключі з однаковим значенням цієї цифри об’єднуються в одну групу. Ключі в групі розташовуються в порядку їхнього надходження. Після того, як вся початкова послідовність розподілена по групах, групи розташовуються в порядку зростання пов’язаних з групами цифр. Процес повторюється для другої цифри і т.д., поки не будуть вичерпані значущі цифри в ключі. Основа системи числення P може бути будь-якою при цьому потрібно P груп.

Порядок алгоритму якісно лінійний – O(N), для сортування потрібно D*N операцій аналізу цифри. Проте, в такій оцінці порядку не враховується ряд обставин.

По-перше, операція виділення значущої цифри буде простою і швидкою тільки при P=2, для інших систем числення ця операція може вимагати значно більше часу, ніж операція порівняння.

По-друге, при оцінці алгоритму не враховуються затрати часу і пам’яті на створення і ведення груп. Розміщення груп в статичній робочій пам’яті вимагає пам’яті для P*N елементів, оскільки в граничному випадку всі елементи можуть потрапити в якусь одну групу. Якщо ж формувати групи усередині тієї ж послідовності за принципом обмінних алгоритмів, то виникає необхідність перерозподілу послідовності між групами і всі проблеми і недоліки, властиві алгоритмам включення. Найбільш раціональним є формування груп у вигляді зв’язних списків з динамічним виділенням пам’яті.

Алгоритм швидкого сортування Хоара відноситься до розподільних і забезпечує показники ефективності O(N*log2(N)) навіть при якнайгіршому початковому розподілі.

Використовується два індекси з початковими значеннями початку і кінця множини відповідно. Ключ початку порівнюється з ключем кінця. Якщо ключі задовольняють критерію впорядкованості, то індекс кінця зменшується на 1 і проводиться наступне порівняння. Якщо ключі не задовольняють критерію, то записи міняються місцями. При цьому індекс кінця фіксується і починає мінятися індекс початку (збільшуватися на 1 після кожного порівняння). Після наступної

перестановки фіксується початок і починає змінюватися кінець і т.д. Прохід закінчується, коли індекси стають рівними. Запис, що знаходиться на позиції зустрічі індексів, стоїть на своєму місці в послідовності. Цей запис ділить послідовність на дві підмножини. Всі записи, розташовані ліворуч від неї мають ключі, менші ніж ключ цього запису, всі записи праворуч – більші. Той же самий алгоритм застосовується до лівої підмножини, а потім до правої. Записи підмножини розподіляються на дві менші підмножини і так далі. Розподіл закінчується, коли отримана підмножина буде складатися з єдиного елемента – така підмножина вже є впорядкованою.

4.5. Сортування злиттям.

Алгоритми сортування злиттям, як правило, мають порядок O(N*log2(N)), але відрізняються від інших алгоритмів більшою складністю і вимагають великої кількості пересилок. Алгоритми злиття застосовуються в основному, як складова частина зовнішнього сортування.

При сортуванні попарним злиттям вхідна множина розглядається, як послідовність підмножин, кожна з яких складається з єдиного елемента і, отже, є вже впорядкованим. На першому проході кожні дві сусідні одноелементних множини зливаються в одну двоелементну впорядковану множину. На другому проході двоелементні множини зливаються в 4-елементні впорядковані множини і т.д. Врешті-решт отримують одну велику впорядковану множину.

Самою найважливішою частиною алгоритму є злиття двох впорядкованих множин. Цю частину алгоритму опишемо більш детально.

1. Початкові установки. Визначити довжини першої і другої початкових множин – l1 і l2 відповідно. Встановити індекси поточних елементів в початковій множині i1 і i2 в 0. Встановити індекс в вихідній множині j=1.

2. Цикл злиття. Виконувати крок 3 до тих пір, поки i1<=l1 і i2<=l2.

3. Порівняння. Порівняти ключ i1-го елемента з першої початкової множини з ключем i2-го елемента з другої початкової множини. Якщо ключ елемента з 1-ої множини менший, то записати i1-тий елемент з 1-ої множини на j-те місце в

вихідній множині і збільшити i1 на 1. Інакше – записати i2-тий елемент з 2-ої множини на j-те місце в вихідній множині і збільшити i2 на 1. Збільшити j на 1.

4. Виведення залишків. Якщо i1<=l1, то переписати частину 1-ої початкової множини від i1 до l1 включно в вихідну множину. Інакше – переписати частину 2-ої початкової множини від i2 до l2 включно в вихідну множину.

4.6. Рандомізація

В деяких програмах потрібно виконання операцій, протилежних сортуванню. Отримавши множину елементів, програма повинна розмістити їх у випадковому порядку. Рандомізацію нескладно виконати, використовуючи алгоритм, подібний на сортування вибіркою.

Для кожного розміщення в множині, алгоритм випадковим чином вибирає елемент, який повинен його зайняти з тих, які ще не були розміщені на своєму місці. Потім цей елемент міняється місцями з елементом, який, знаходиться на цій позиції. Так як алгоритм заповнює кожну позицію лише один раз, його складність – O(N).

Нескладно показати, що імовірність того, що елемент виявиться на якійнебудь позиції, рівна 1/N. Оскільки елемент може виявитися на будь-якій позиції з однаковою імовірністю, цей алгоритм дійсно приводить до випадкового розміщення елементів.

Результат рандомізації залежить від того, наскільки ефективним є генератор випадкових чисел. Для даного алгоритми не важливий початковий порядок розміщення елементів. Якщо необхідно неодноразово рандомізувати множину елементів, немає необхідності її попередньо сортувати.

5. ПРИНЦИПИ SOLID

Як розштфровується SOLID:

- S: Single Responsibility Principle (Принцип одної відповідальності).
- O: Open-Closed Principle (Принцип відкриитості-закритості).
- L: Liskov Substitution Principle (Принцип підстановки Барбари Лісков).
- I: Interface Segregation Principle (Принцип розділення інтерфейса).
- D: Dependency Inversion Principle (Принцип інверсії залежностей).

Зараз ми розглянемо ці принципи на схематичних прикладах. Зверніть увагу на те, що головна мета прикладів полягає в тому, щоб допомогти читачеві зрозуміти принципи SOLID, дізнатися, як їх застосовувати і як дотримуватися їх, проектуючи додатки. Автор матеріалу не прагнув до того, щоб вийти на працюючий код, який можна було б використовувати в реальних проектах.

1. Принцип єдиної відповідальності:

Клас повинен бути відповідальний лише за щось одне. Якщо клас відповідає за вирішення декількох завдань, його підсистеми, що реалізують рішення цих задач, виявляються пов'язаними один з одним. Зміни в одній такій підсистемі ведуть до змін в іншій.

Зверніть увагу на те, що цей принцип можна застосувати не тільки до класів, а й до компонентів програмного забезпечення в більш широкому сенсі.

Клас Animal, представлений тут, описує якусь тварину. Цей клас порушує принцип єдиної відповідальності. Як саме порушується цей принцип?

Відповідно до принципу єдиної відповідальності клас повинен вирішувати лише якусь одну задачу. Він же вирішує дві, займаючись роботою зі сховищем даних в методі saveAnimal і маніпулюючи властивостями об'єкта в конструкторі і в методі getAnimalName.

Якщо зміниться порядок роботи зі сховищем даних, що використовуються додатком, то доведеться вносити зміни в усі класи, які працюють зі сховищем. Така архітектура не відрізняється гнучкістю, зміни одних підсистем зачіпають інші, що нагадує ефект доміно.

Для того щоб привести вищенаведений код у відповідність до принципу єдиної відповідальності, створимо ще один клас, єдиним завданням якого є робота зі сховищем, зокрема - збереження в ньому об'єктів класу Animal.

Ось що з цього приводу говорить Стів Фентон: «Проектуючи класи, ми повинні прагнути до того, щоб об'єднувати споріднені компоненти, тобто такі, зміни в яких відбуваються за одними і тими ж причинами. Нам слід намагатися розділяти компоненти, зміни в яких викликають різні причини ».

Правильне застосування принципу єдиної відповідальності призводить до високого ступеня зв'язності елементів всередині модуля, тобто до того, що завдання, які вирішуються всередині нього, добре відповідають його головної мети.

2. Принцип відкритості-закритості:

Ми хочемо перебрати список тварин, кожне з яких представлено об'єктом класу Animal, і дізнатися про те, які звуки вони видають. Уявімо, що ми вирішуємо це завдання за допомогою функції AnimalSounds.

Найголовніша проблема такої архітектури полягає в тому, що функція визначає те, який звук видає ту чи іншу тварину, аналізуючи конкретні об'єкти. Функція

AnimalSound не відповідає принципу відкритості-закритості, так як, наприклад, при появі нових видів тварин, нам, для того, щоб з її допомогою можна було б дізнаватися звуки, що видаються ними, доведеться її змінити.

Як бачите, при додаванні в масив нового тваринного доведеться доповнювати код функції. Приклад це дуже простий, але якщо подібна архітектура використовується в реальному проекті, функцію доведеться постійно розширювати, додаючи в неї нові вирази if.

Можна помітити, що у класу Animal тепер є віртуальний метод makeSound. При такому підході потрібно, щоб класи, призначені для опису конкретних тварин, розширювали б клас Animal і реалізовували б цей метод.

В результаті у кожного класу, що описує тваринного, буде власний метод makeSound, а при переборі масиву з тваринами в функції AnimalSound досить буде викликати цей метод для кожного елемента масиву.

Якщо тепер додати в масив об'єкт, що описує нова тварина, функцію AnimalSound змінювати не треба. Ми привели її у відповідність до принципу відкритості-закритості.

3. Принцип підстановки Барбари Лісков:

Мета цього принципу полягають в тому, щоб класи-спадкоємці могли б використовуватися замість батьківських класів, від яких вони утворені, не порушуючи роботу програми. Якщо виявляється, що в коді перевіряється тип класу, значить принцип підстановки порушується.

Розглянемо застосування цього принципу, повернувшись до прикладу з класом Animal. Напишемо функцію, призначену для повернення інформації про кількість кінцівок тварини.

В результаті, наприклад, при зверненні до методу LegCount для екземпляра класу Lion здійснюється виклик методу, реалізованого в цьому класі, і повертається саме те, що можна очікувати від виклику подібного методу.

Тепер функції AnimalLegCount не потрібно знати про те, об'єкт якого саме підкласу класу Animalona обробляє для того, щоб дізнатися відомості про

кількість кінцівок у тварини, представленого цим об'єктом. Функція просто викликає метод LegCount класу Animal, так як підкласи цього класу повинні реалізовувати цей метод для того, щоб їх можна було б використовувати замість нього, не порушуючи правильність роботи програми.

4. Принцип поділу інтерфейсу:

Цей принцип спрямований на усунення недоліків, пов'язаних з реалізацією великих інтерфейсів.

Він описує методи для малювання кіл (drawCircle), квадратів (drawSquare) і прямокутників (drawRectangle). В результаті класи, що реалізують цей інтерфейс і представляють окремі геометричні фігури, такі, як коло (Circle), квадрат (Square) і прямокутник (Rectangle), повинні містити реалізацію всіх цих методів.

Принцип поділу інтерфейсу застерігає нас від створення інтерфейсів, подібних Shape з нашого прикладу. Клієнти (у нас це класи Circle, Square і Rectangle) не повинні реалізовувати методи, які їм не потрібно використовувати. Крім того, цей принцип вказує на те, що інтерфейс повинен вирішувати лише якусь одну задачу (в цьому він схожий на принцип єдиної відповідальності), тому все, що виходить за рамки цього завдання, має бути винесено в інший інтерфейс або інтерфейси.

5. Принцип інверсії залежностей:

- Модулі верхніх рівнів не повинні залежати від модулів нижніх рівнів. Обидва типи модулів повинні залежати від абстракцій.

- Абстракції не повинні залежати від деталей. Деталі повинні залежати від абстракцій.

В процесі розробки програмного забезпечення існує момент, коли функціонал додатка перестає поміщатися в рамках одного модуля. Коли це відбувається, нам доводиться вирішувати проблему залежностей модулів. В результаті, наприклад, може виявитися так, що високорівневі компоненти залежать від низькорівневих компонентів.

Тут клас Http є високорівнева компонент, а XMLHttpService - низькорівневий. Така архітектура порушує пункт А принципу інверсії залежностей: «Модулі верхніх рівнів не повинні залежати від модулів нижніх рівнів. Обидва типи модулів повинні залежати від абстракцій ».

Клас Http вимушено залежить від класу XMLHttpService. Якщо ми вирішимо змінити механізм, який використовується класом Http для взаємодії з мережею - скажімо, це буде Node.js-сервіс або, наприклад, сервіс-заглушка, застосовуваний для цілей тестування, нам доведеться відредагувати всі екземпляри класу Http, змінивши відповідний код. Це порушує принцип відкритості-закритості.

Як можна помітити, тут високорівневі і низькорівневі модулі залежать від абстракцій. Клас Http (високорівнева модуль) залежить від інтерфейсу Connection (абстракція). Класи XMLHttpService, NodeHttpService і MockHttpService (низькорівневі модулі) також залежать від інтерфейсу Connection.

Крім того, варто відзначити, що дотримуючись принципу інверсії залежностей, ми дотримуємося і принцип підстановки Барбари Лісков. А саме, виявляється, що типи XMLHttpService, NodeHttpService і MockHttpService можуть служити заміною базового типу Connection.

ВИСНОВОК

Навчальна програма була розроблена на мові програмування Java. В даний час об'єктно-орієнтоване програмування є основним напрямом розвитку програмування взагалі, а мова Java є повністю об'єктною мовою. А наявність засобів строгої перевірки типів, орієнтація на роботу з комп'ютерними мережами, перенесення на рівні виконуваного коду і підтримка платформонезалежного коду, а також заборону прямого звернення до апаратури забезпечили виконання більшості вимог, що пред'являлися до мови прикладного програмування.

В ході курсової роботи був закріплений досвід роботи з мовою Java, закріплені навики роботи в середовищі розробки IDEA.

Здобуто та закріплено навички написання черг з пріорітетом , також вивчені основні алгоритми написання коду.

Дотримані основні принципи ООП такі як : інкапсуляція, наслідування та поліморфізм.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Дейтел Х.М., Дейтел П.Дж. Как программировать на Java. Книга 1. Основы программирования – М.: Бином, 2006. – 848 с.

2. Дейтел Х.М., Дейтел П.Дж. Как программировать на Java. Книга 2. Файлы, сети, базы данных. – М.: Бином, 2006. – 672 с.

3. Хорстманн К. С., Корнелл Г. Java 2. Библиотека профессионала, том 1. Основы, 7-е изд. – М. : Издательский дом "Вильямс", 2006. – 896 стр. с ил.

4. Java Tutorials – http://docs.oracle.com/javase/tutorial (англ.)

5. Метод хорд – Вікіпедія http://uk.wikipedia.org/wiki/ Метод_хорд

6. Шилдт Г., Холмс Д. Искусство программирования на Java. – М.: Издательский дом "Вильямс", 2005. – 336 стр.

7. Эккель Б. Философия Java. Библиотека программиста. 3-е изд. – СПб.: Питер, 2003. – 976 с.

8. Копитко М.Ф., Іванків К.С. Основи програмування мовою Java: Тексти лекцій. – Львів: Видавничий центр ЛНУ ім. Івана Франка, 2002. – 83 с.

9 Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес «Паттерны объектно-ориентированного проектирования» Питер, 2020. – 448 стр.

10 Бэзинс, Барт. Java для начинающих : объектно-ориентированный подход / Барт Бэзинс, Эйми Бэкил, Зеппе ванден Бруке ; пер. А. Ананич, Е. Зазноба, А. Колышкин, А. Тумаркин. – Санкт-Петербург [и др.] : Питер, 2018. – 688 с.

11 Буч Г., РАМБО Д., Якобсон И. Язык UML. Руководство пользователя. 2-е изд.: ПЕР. с англ. Мухин Н.-М.: ДМК Пресс,2007. -496с.:ил.

12 Мацяшек, Лешек А. Анализ и проэктирование информационных систем с помощью UML 2.0, 3-е изд.: Пер.сангл.-М.: ООО «И.Д.Вильямс»2008.-816с.:ил

13 Шилдт, Герберт. Java 8. Полное руководство; 9-е изд.: Пер. с англ. - М.: ООО "И.Д. Вильяме", 2015. – 1376 с.

14 Джеймс Гослинг, Билл Джой, Гай Стил, Гилад Брача, Алекс Бакли. Язык программирования Java SE 8. Подробное описание, 5-е издание. – М.: "Вильямс", 2015. – 672 с.

15 Блох, Джошуа. Java: эффективное программирование, 3-е изд. : Пер. с англ. – СПб. : ООО "Диалектика", 2019. – 464 с.

16 Хорстманн, Кей С. Java. Библиотека профессионала, том 1. Основы. 10-е изд.: Пер. с англ. – М .: ООО "И.Д. Вильямс", 2016. - 864 с.

17 Леоненков А. В. Самоучитель UML2.-СПб.: БХВ-Петербург, 2007.-576с.:ил.

18 https://uk.m.wikipedia.org/wiki/%D0%A7%D0%B5%D1%80%D0%B3%D0%B0_%D0%B7_%D0%BF%D1%80%D1%96%D0%BE%D1%80%D0%B8%D1%82%D0%B5%D1%82%D0%BE%D0%BC

19 https://living-sun.com/uk/java/491776-when-adding-values-to-a-priority-queue-when-exactly-are-the-values-sorted-java-sorting-comparator-priority-queue.html

20 https://coderoad.ru/2702913/%D0%A0%D0%B5%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F-%D0%9F%D1%80%D0%B8%D0%BE%D1%80%D0%B8%D1%82%D0%B5%D1%82%D0%BD%D0%BE%D0%B9-%D0%9E%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D0%B8-Java

21 https://vertex-academy.com/tutorials/ru/queue-java-primer/

22 https://www.cyberforum.ru/java-j2se/thread190700.html

23 http://ukr-technologies.blogspot.com/2020/04/priorityqueue-java.html

24 https://javadevblog.com/primer-ispol-zovaniya-java-priority-queue-priorityqueue.html

25 https://hackit-ukraine.com/302-priority-queues-in-java-explained-with-examples

26 https://uk.myservername.com/java-priority-queue-tutorial-implementation-examples

ДОДАТОК А

```
import Service.IPriority;
import Service.PriorQueue;

public class Program {

    public static void main (String[] args) {
        IPriority priorQueue = new PriorQueue();
        priorQueue.fillQueue();
        priorQueue.size();
        priorQueue.print();
        priorQueue.back();
        priorQueue.empty();
        priorQueue.pop();
        priorQueue.push();
        priorQueue.find();

    }
```

```
}
```

ДОДАТОК Б

```
package Service;

import models.Students;

import java.util.Comparator;
import java.util.PriorityQueue;

public class PriorQueue implements IPriority{


    static PriorityQueue<Students> pq = new PriorityQueue<Students>();


        //Створення студентів
        @Override
        public void fillQueue() {
            pq.add(new Students("Jack", 5));
```

```
    pq.add(new Students("Georg", 2));
    pq.add(new Students("David", 1));
    pq.add(new Students("Barbara", 4));
    pq.add(new Students("Ricky", 3));

}

//Вивід
@Override
public void print() {
    while (!pq.isEmpty()) {
        System.out.println("\t" + pq.poll());
    }
}

//back
@Override
public void back() {
    pq.add(new Students("Ste", 2));
    pq.add(new Students("Joo", 1));
    pq.add(new Students("Scott", 6));

    for(int i = 0; i< pq.size(); i++) pq.remove();
    System.out.println("Last element " + pq.peek());
}

//size
@Override
public void size(){
    System.out.println("Size: " + pq.size());
```

```
}

//empty
@Override
public void empty(){
    if (pq.isEmpty()){
        System.out.println("\nFalse");
    } else System.out.println("\nTrue");

}

//pop (Remove max and mid element)
@Override
public void pop(){
    pq.add(new Students("Joo", 1));
    pq.add(new Students("Nick", 5));
    pq.add(new Students("Ste", 2));

    for(int i = pq.size(); i <= pq.size(); i++) pq.remove();
    System.out.println("\nAfter remove first and last element: " + pq.poll());
}

//push
@Override
public void push(){

    if (!pq.isEmpty()){
        System.out.println("\nAfter add : ");
        pq.add(new Students("Olya",7));
        }
```

```
        while (!pq.isEmpty()) {
            System.out.println("\t" + pq.poll());
        }
    }

    //find
    @Override
    public void find(){
        pq.add(new Students("Jack", 5));
        pq.add(new Students("Georg", 2));
        pq.add(new Students("David", 1));
        pq.add(new Students("Barbara", 4));
        pq.add(new Students("Ricky", 3));

        System.out.println("\nJack is here?");
        if (!pq.contains("Jack")){
            System.out.println("YES!!!");
        }else System.out.println("Can't find");

    }
}
```

ДОДАТОК В

```
package Service;

public interface IPriority {
    void fillQueue();
    void print();
    void back();
    void size();
    void empty();
    void pop();
    void push();
    void find();
}
```

ДОДАТОК Г

```
package models;
import java.util.*;


public class Students implements Comparable<Students>  {

    private String name;
    private int course;

    public Students(String name, int course) {
        this.name = name;
        this.course = course;
    }

    public Students() {
    }
```

```
public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public int getCourse() {
    return course;
}

public void setCourse(int course) {
    this.course = course;
}

@Override
public String toString() {
    return "Students{" +
            "name='" + name + '\'' +
            ", course=" + course +
            '}';
}

@Override
public int compareTo(Students other) {
    return this.course-other.course;
}

@Override
```

```
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Students students = (Students) o;
        return course == students.course ;
    }

    @Override
    public int hashCode() {
        return Objects.hash(course);
    }
}
```