**Design Principal and Patterns**

Q3>

Code:

class Computer{

private final String CPU;

private final String RAM;

private final String storage;

private Computer(Builder builder){

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

}

public String getCPU(){

return this.CPU;

}

public String getRAM(){

return this.RAM;

}

public String getStorage(){

return this.storage;

}

public String toString() {

return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", Storage=" + storage;

}

public static class Builder{

private String CPU;

private String RAM;

private String storage;

public Builder setCPU(String CPU){

this.CPU = CPU;

return this;

}

public Builder setRAM(String RAM){

this.RAM = RAM;

return this;

}

public Builder setStorage(String storage){

this.storage = storage;

return this;

}

public Computer build(){

return new Computer(this);

}

}

}

public class BuilderPatternExample

{

public static void main(String[] args){

Computer basicComputer = new Computer.Builder()

.setCPU("Intel i3")

.setRAM("4GB")

.setStorage("500GB HDD")

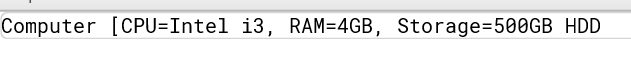
.build();

System.out.println(basicComputer);

}

}

Output



Q4>

Code

interface PaymentProcessor{

void processPayment(double amount);

}

class GooglePayGateway{

public void sendPayment(double amount){

System.out.println("Processing GooglePay payment of $"+amount);

}

}

class PaytmGateway{

public void makePayment(double amount){

System.out.println("Processing Paytm payment of $"+amount);

}

}

class GooglePayAdapter implements PaymentProcessor{

private GooglePayGateway googlePayGateway;

public GooglePayAdapter(GooglePayGateway googlePayGateway){

this.googlePayGateway = googlePayGateway;

}

public void processPayment(double amount){

googlePayGateway.sendPayment(amount);

}

}

class PaytmAdapter implements PaymentProcessor{

private PaytmGateway paytmGateway;

public PaytmAdapter(PaytmGateway paytmGateway){

this.paytmGateway = paytmGateway;

}

public void processPayment(double amount){

paytmGateway.makePayment(amount);

}

}

public class AdapterPatternExample

{

public static void main(String[] args){

PaymentProcessor googlePayProcessor = new GooglePayAdapter(new GooglePayGateway());

googlePayProcessor.processPayment(120.50);

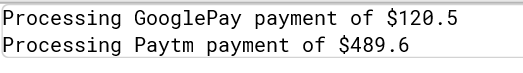
PaymentProcessor paytmProcessor = new PaytmAdapter(new PaytmGateway());

paytmProcessor.processPayment(489.60);

}

}

Output



Q5>

Code:

interface Notifier {

void send(String message);

}

class EmailNotifier implements Notifier {

public void send(String message) {

System.out.println("Sending Email: " + message);

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier wrappedNotifier;

public NotifierDecorator(Notifier notifier) {

this.wrappedNotifier = notifier;

}

public void send(String message) {

wrappedNotifier.send(message);

}

}

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

System.out.println("Sending SMS: " + message);

}

}

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

System.out.println("Sending Slack message: " + message);

}

}

public class DecoraterPatternExample

{

public static void main(String[] args) {

Notifier notifier = new EmailNotifier();

notifier = new SMSNotifierDecorator(notifier);

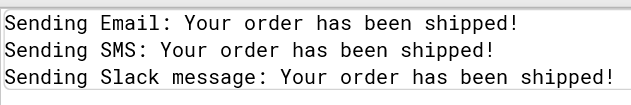
notifier = new SlackNotifierDecorator(notifier);

notifier.send("Your order has been shipped!");

}

}

Output



Q6>

Code:

interface Image {

void display();

}

class RealImage implements Image {

private String fileName;

public RealImage(String fileName) {

this.fileName = fileName;

loadFromRemoteServer();

}

private void loadFromRemoteServer() {

System.out.println("Loading image from remote server: " + fileName);

}

public void display() {

System.out.println("Displaying image: " + fileName);

}

}

class ProxyImage implements Image {

private String fileName;

private RealImage realImage;

public ProxyImage(String fileName) {

this.fileName = fileName;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(fileName);

}

realImage.display();

}

}

public class ProxyPatternExample

{

public static void main(String[] args) {

Image image1 = new ProxyImage("photo1.jpg");

Image image2 = new ProxyImage("photo2.jpg");

System.out.println("--- First time displaying photo1 ---");

image1.display();

System.out.println("--- First time displaying photo2 ---");

image2.display();

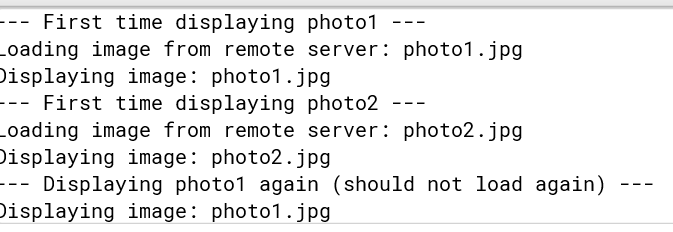
System.out.println("--- Displaying photo1 again (should not load again) ---");

image1.display();

}

}

Output



Q8>

Code:

interface PaymentStrategy {

void pay(double amount);

}

class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

public CreditCardPayment(String cardNumber) {

this.cardNumber = cardNumber;

}

public void pay(double amount) {

System.out.println("Paid $" + amount + " using Credit Card: " + cardNumber);

}

}

class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

public void pay(double amount) {

System.out.println("Paid $" + amount + " using PayPal account: " + email);

}

}

class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void executePayment(double amount) {

if (strategy == null) {

System.out.println("Payment strategy not set.");

} else {

strategy.pay(amount);

}

}

}

public class StrategyPatternExample

{

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-3456"));

context.executePayment(250.00);

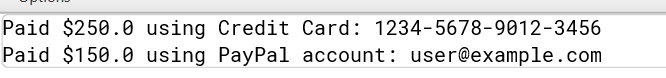
context.setPaymentStrategy(new PayPalPayment("user@example.com"));

context.executePayment(150.00);

}

}

Output



Q10>

Code:

class Student {

private String id;

private String name;

private String grade;

public Student(String id, String name, String grade) {

this.id = id;

this.name = name;

this.grade = grade;

}

public String getId() { return id; }

public void setId(String id) { this.id = id; }

public String getName() { return name; }

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

public String getGrade() { return grade; }

public void setGrade(String grade) { this.grade = grade; }

}

class StudentView {

public void displayStudentDetails(String id, String name, String grade) {

System.out.println("Student Details:");

System.out.println("ID: " + id);

System.out.println("Name: " + name);

System.out.println("Grade: " + grade);

}

}

class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void setStudentName(String name) {

model.setName(name);

}

public String getStudentName() {

return model.getName();

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public String getStudentGrade() {

return model.getGrade();

}

public void updateView() {

view.displayStudentDetails(model.getId(), model.getName(), model.getGrade());

}

}

public class MVCPatternExample

{

public static void main(String[] args) {

Student student = new Student("101", "Alice", "A");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentName("Alice Johnson");

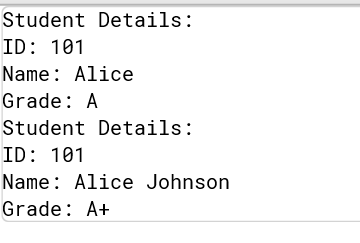
controller.setStudentGrade("A+");

controller.updateView();

}

}

Output



Q11>

Code:

interface CustomerRepository {

String findCustomerById(String id);

}

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

return "Customer[id=" + id + ", name=John Doe]";

}

}

class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void displayCustomer(String id) {

String customer = repository.findCustomerById(id);

System.out.println("Found: " + customer);

}

}

public class DependencyInjectionExample

{

public static void main(String[] args) {

CustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

service.displayCustomer("C123");

}

}

Output



**Data Structures and Algorithm**

Q1>

Step1.

Explain why data structures and algorithms are essential in handling large inventories

Ans: In an inventory management system that handles large inventories, data structures and algorithms play a crucial role in ensuring the system is efficient, scalable, and reliable. As the number of products grows, it becomes increasingly important to store and retrieve data quickly so that operations like adding, searching, updating, or deleting product records can be done without delays. A well-chosen data structure helps minimize the time it takes to perform these operations, even when dealing with thousands or millions of items. Similarly, good algorithms ensure that processes such as searching for a product by ID or updating stock levels run in optimal time, which is vital for smooth warehouse operations, real-time updates, and decision making.

Discuss the types of data structures suitable for this problem.

Ans: For this problem, HashMap is one of the most suitable data structures because it allows us to store product records using unique keys (such as product IDs) and provides fast average time complexity of O(1) for search, insert, update, and delete operations. This means we can quickly locate any product based on its ID, which is a common requirement in inventory systems. If we wanted to maintain an ordered view of products (such as sorting by name or price), we could additionally use structures like a TreeMap or a combination of a HashMap and a sorted list. For cases where indexed access or iteration over the list of products in insertion order is important, we might consider an ArrayList, but its search and delete operations are slower (O(n)) and therefore not ideal for large inventories. The choice of data structure ultimately depends on the access patterns and performance requirements of the system.

Code:

import java.util.HashMap;

import java.util.Map;

class Product {

private String productId;

private String productName;

private int quantity;

private double price;

public Product(String productId, String productName, int quantity, double price) {

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

// Getters and setters

public String getProductId() { return productId; }

public String getProductName() { return productName; }

public int getQuantity() { return quantity; }

public double getPrice() { return price; }

public void setProductName(String productName) { this.productName = productName; }

public void setQuantity(int quantity) { this.quantity = quantity; }

public void setPrice(double price) { this.price = price; }

@Override

public String toString() {

return "[" + productId + "] " + productName + " - Qty: " + quantity + " Price: $" + price;

}

}

class Inventory {

private Map<String, Product> products = new HashMap<>();

public void addProduct(Product product) {

products.put(product.getProductId(), product);

}

public void updateProduct(String productId, String name, int quantity, double price) {

Product product = products.get(productId);

if (product != null) {

product.setProductName(name);

product.setQuantity(quantity);

product.setPrice(price);

} else {

System.out.println("Product not found: " + productId);

}

}

public void deleteProduct(String productId) {

products.remove(productId);

}

public void displayAllProducts() {

for (Product p : products.values()) {

System.out.println(p);

}

}

public Product findProductById(String productId) {

return products.get(productId);

}

}

public class ManageInventory

{

public static void main(String[] args) {

Inventory inventory = new Inventory();

// Add products

inventory.addProduct(new Product("P001", "Laptop", 10, 999.99));

inventory.addProduct(new Product("P002", "Mouse", 50, 19.99));

// Display products

System.out.println("--- All Products ---");

inventory.displayAllProducts();

// Update product

inventory.updateProduct("P002", "Wireless Mouse", 45, 24.99);

// Display after update

System.out.println("--- After Update ---");

inventory.displayAllProducts();

// Delete product

inventory.deleteProduct("P001");

// Display after delete

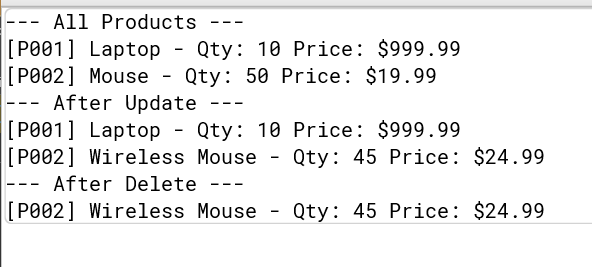
System.out.println("--- After Delete ---");

inventory.displayAllProducts();

}

}

Output



Analysis:

| **Operation** | **Data Structure** | **Complexity** |
| --- | --- | --- |
| Add | HashMap | O(1) average |
| Update | HashMap | O(1) average (find + update) |
| Delete | HashMap | O(1) average |
| Search by ID | HashMap | O(1) average |

Q3>

Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).

Ans: **Bubble Sort**:  
This is a simple comparison-based algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. It continues until no more swaps are needed. It’s easy to implement but inefficient for large datasets, with a time complexity of **O(n²)**.

**Insertion Sort**:  
This algorithm builds the sorted list one element at a time. It takes each new element and inserts it into its proper position relative to those already sorted. Like bubble sort, it has **O(n²)** time complexity in the worst case, but it performs well on nearly sorted data.

**Quick Sort**:  
A highly efficient, divide-and-conquer algorithm. It picks a *pivot* element, partitions the list into two sublists (elements less than pivot and elements greater than pivot), and recursively sorts the sublists. On average, it has **O(n log n)** time complexity, though in the worst case it can degrade to **O(n²)** (e.g., if pivot selection is poor).

**Merge Sort**:  
Another divide-and-conquer algorithm that splits the list in half, recursively sorts both halves, and then merges them back together in sorted order. Its time complexity is **O(n log n)** in all cases, but it requires extra space for merging.

Code:

class Order {

private String orderId;

private String customerName;

private double totalPrice;

public Order(String orderId, String customerName, double totalPrice) {

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

public String getOrderId() { return orderId; }

public String getCustomerName() { return customerName; }

public double getTotalPrice() { return totalPrice; }

@Override

public String toString() {

return "[" + orderId + "] " + customerName + " - $" + totalPrice;

}

}

class Sorter {

public static void bubbleSort(Order[] orders) {

int n = orders.length;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (orders[j].getTotalPrice() > orders[j + 1].getTotalPrice()) {

Order temp = orders[j];

orders[j] = orders[j + 1];

orders[j + 1] = temp;

}

}

}

}

public static void quickSort(Order[] orders, int low, int high) {

if (low < high) {

int pi = partition(orders, low, high);

quickSort(orders, low, pi - 1);

quickSort(orders, pi + 1, high);

}

}

private static int partition(Order[] orders, int low, int high) {

double pivot = orders[high].getTotalPrice();

int i = (low - 1);

for (int j = low; j < high; j++) {

if (orders[j].getTotalPrice() < pivot) {

i++;

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

Order temp = orders[i + 1];

orders[i + 1] = orders[high];

orders[high] = temp;

return i + 1;

}

}

public class SortingAlgo

{

public static void main(String[] args) {

Order[] orders = {

new Order("O001", "Alice", 250.0),

new Order("O002", "Bob", 150.0),

new Order("O003", "Charlie", 300.0)

};

System.out.println("--- Before sorting ---");

for (Order o : orders) System.out.println(o);

// Bubble sort

Sorter.bubbleSort(orders);

System.out.println("--- After Bubble Sort ---");

for (Order o : orders) System.out.println(o);

// Reset for quick sort

orders = new Order[] {

new Order("O001", "Alice", 250.0),

new Order("O002", "Bob", 150.0),

new Order("O003", "Charlie", 300.0)

};

Sorter.quickSort(orders, 0, orders.length - 1);

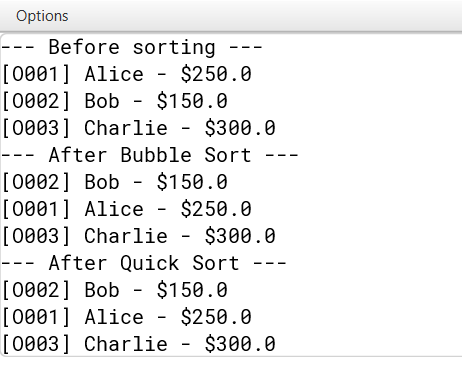
System.out.println("--- After Quick Sort ---");

for (Order o : orders) System.out.println(o);

}

}

Output



Analysis

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |  |
| --- | --- | --- | --- | --- |
| Bubble Sort | O(n) | O(n²) | O(n²) |  |
| Quick Sort | O(n log n) | O(n log n) | O(n²) |  |

Quick Sort is preferred because it sorts large datasets much faster on average by efficiently partitioning data.

Q4>

Explain how arrays are represented in memory and their advantages.

Ans: In memory, an array is represented as a contiguous block of memory where each element is stored at consecutive memory addresses. When an array is created, the system allocates a fixed-size block of memory large enough to hold all its elements.

**Advantages of arrays**:

* Direct (random) access: You can access any element in O(1) time because you can compute its address directly.
* Efficient memory layout: Since arrays are contiguous, they have good cache locality, making them fast for sequential access and traversal.
* Simple to use: Syntax and logic for arrays are straightforward.

Code:

class Employee {

private String employeeId;

private String name;

private String position;

private double salary;

public Employee(String employeeId, String name, String position, double salary) {

this.employeeId = employeeId;

this.name = name;

this.position = position;

this.salary = salary;

}

public String getEmployeeId() { return employeeId; }

public String getName() { return name; }

public String getPosition() { return position; }

public double getSalary() { return salary; }

@Override

public String toString() {

return "[" + employeeId + "] " + name + " - " + position + " ($" + salary + ")";

}

}

class EmployeeManager {

private Employee[] employees;

private int size = 0;

public EmployeeManager(int capacity) {

employees = new Employee[capacity];

}

public void addEmployee(Employee emp) {

if (size < employees.length) {

employees[size++] = emp;

} else {

System.out.println("Employee array is full!");

}

}

public Employee searchEmployee(String employeeId) {

for (int i = 0; i < size; i++) {

if (employees[i].getEmployeeId().equals(employeeId)) {

return employees[i];

}

}

return null;

}

public void deleteEmployee(String employeeId) {

for (int i = 0; i < size; i++) {

if (employees[i].getEmployeeId().equals(employeeId)) {

for (int j = i; j < size - 1; j++) {

employees[j] = employees[j + 1];

}

employees[size - 1] = null;

size--;

System.out.println("Deleted employee: " + employeeId);

return;

}

}

System.out.println("Employee not found: " + employeeId);

}

public void traverseEmployees() {

for (int i = 0; i < size; i++) {

System.out.println(employees[i]);

}

}

}

public class TestEmployee

{

public static void main(String[] args) {

EmployeeManager manager = new EmployeeManager(5);

manager.addEmployee(new Employee("E001", "Alice", "Manager", 75000));

manager.addEmployee(new Employee("E002", "Bob", "Developer", 60000));

manager.addEmployee(new Employee("E003", "Charlie", "Analyst", 55000));

System.out.println("--- All Employees ---");

manager.traverseEmployees();

System.out.println("--- Search for E002 ---");

Employee e = manager.searchEmployee("E002");

System.out.println(e != null ? e : "Not found");

System.out.println("--- Delete E001 ---");

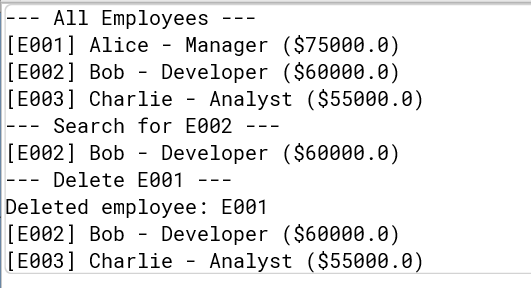
manager.deleteEmployee("E001");

manager.traverseEmployees();

}

}

Output



Analysis:

| **Operation** | **Time Complexity** |  |
| --- | --- | --- |
| Add | O(1) |  |
| Search | O(n) |  |
| Traverse | O(n) |  |
| Delete | O(n) |  |

Limitations of arrays

* Fixed size: You must know the maximum number of employees up front, or resize manually.
* Costly deletion: Removing an element requires shifting subsequent elements.
* Wasted space: If the array has unused capacity, memory is wasted.

Q5>

Explain the different types of linked lists (Singly Linked List, Doubly Linked List).

Ans: **Singly Linked List**:  
Each node points to the next node.  
Structure: data → next → null  
Simple, less memory per node, but no backward traversal.

**Doubly Linked List**:  
Each node points to both next and previous nodes.  
Structure: prev ← data → next

Allows traversal in both directions but uses more memory per node.

Code:

class Task {

private String taskId;

private String taskName;

private String status;

public Task(String taskId, String taskName, String status) {

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

public String getTaskId() { return taskId; }

public String getTaskName() { return taskName; }

public String getStatus() { return status; }

@Override

public String toString() {

return "[" + taskId + "] " + taskName + " (" + status + ")";

}

}

class TaskNode {

Task task;

TaskNode next;

public TaskNode(Task task) {

this.task = task;

this.next = null;

}

}

class TaskList {

private TaskNode head;

public void addTask(Task task) {

TaskNode newNode = new TaskNode(task);

if (head == null) {

head = newNode;

} else {

TaskNode current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

}

}

public Task searchTask(String taskId) {

TaskNode current = head;

while (current != null) {

if (current.task.getTaskId().equals(taskId)) {

return current.task;

}

current = current.next;

}

return null;

}

public void deleteTask(String taskId) {

if (head == null) return;

if (head.task.getTaskId().equals(taskId)) {

head = head.next;

System.out.println("Deleted task: " + taskId);

return;

}

TaskNode current = head;

while (current.next != null) {

if (current.next.task.getTaskId().equals(taskId)) {

current.next = current.next.next;

System.out.println("Deleted task: " + taskId);

return;

}

current = current.next;

}

System.out.println("Task not found: " + taskId);

}

public void traverseTasks() {

TaskNode current = head;

while (current != null) {

System.out.println(current.task);

current = current.next;

}

}

}

public class TaskListTest

{

public static void main(String[] args) {

TaskList list = new TaskList();

list.addTask(new Task("T001", "Design Database", "Pending"));

list.addTask(new Task("T002", "Implement API", "In Progress"));

list.addTask(new Task("T003", "Write Tests", "Pending"));

System.out.println("--- All Tasks ---");

list.traverseTasks();

System.out.println("--- Search for T002 ---");

Task t = list.searchTask("T002");

System.out.println(t != null ? t : "Not found");

System.out.println("--- Delete T001 ---");

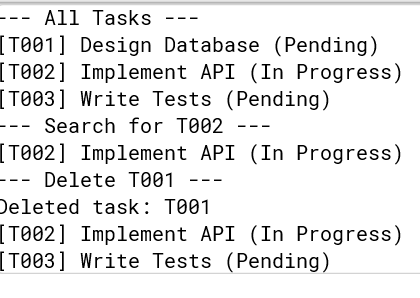
list.deleteTask("T001");

list.traverseTasks();

}

}

Output



Analysis:

| **Operation** | **Time Complexity** |  |
| --- | --- | --- |
| Add (at end) | O(n) |  |
| Search | O(n) |  |
| Traverse | O(n) |  |
| Delete | O(n) |  |

Advantages of linked lists over arrays for dynamic data:

* Dynamic size: Linked lists can grow or shrink at runtime without needing to declare size in advance.
* Efficient insertions/deletions: Inserting or deleting nodes (especially at the start or middle) is faster because no shifting of elements is needed (unlike arrays).
* Memory use: Only uses as much memory as needed (no pre-allocated unused space like arrays).

Q6>

Explain linear search and binary search algorithms.

**Linear Search**

Linear search goes through each element one-by-one until the target is found or the list ends.

Works on unsorted or sorted data.

* Time complexity:
  + Best case: O(1) (target is first element)
  + Worst/average case: O(n)

**Binary Search**

Binary search repeatedly divides a sorted list in half to locate the target.

* Requires the data to be sorted.
* Time complexity:
  + Best case: O(1)
  + Worst/average case: O(log n)

Code

import java.util.Arrays;

import java.util.Comparator;

class Book {

private String bookId;

private String title;

private String author;

public Book(String bookId, String title, String author) {

this.bookId = bookId;

this.title = title;

this.author = author;

}

public String getBookId() { return bookId; }

public String getTitle() { return title; }

public String getAuthor() { return author; }

@Override

public String toString() {

return "[" + bookId + "] " + title + " by " + author;

}

}

class Library {

public static Book linearSearch(Book[] books, String title) {

for (Book book : books) {

if (book.getTitle().equalsIgnoreCase(title)) {

return book;

}

}

return null;

}

public static Book binarySearch(Book[] books, String title) {

int low = 0, high = books.length - 1;

while (low <= high) {

int mid = low + (high - low) / 2;

int cmp = books[mid].getTitle().compareToIgnoreCase(title);

if (cmp == 0) {

return books[mid];

} else if (cmp < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

}

public class LibrarySearch

{

public static void main(String[] args) {

Book[] books = {

new Book("B001", "Java Programming", "John Doe"),

new Book("B002", "Algorithms", "Alice Smith"),

new Book("B003", "Data Structures", "Bob Lee"),

};

// Linear search

System.out.println("--- Linear Search ---");

Book found = Library.linearSearch(books, "Algorithms");

System.out.println(found != null ? found : "Not found");

// Sort for binary search

Arrays.sort(books, Comparator.comparing(Book::getTitle, String.CASE\_INSENSITIVE\_ORDER));

// Binary search

System.out.println("--- Binary Search ---");

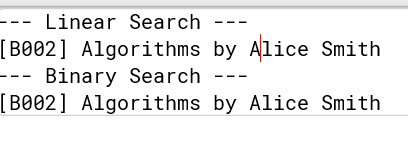
found = Library.binarySearch(books, "Algorithms");

System.out.println(found != null ? found : "Not found");

}

}

Output



Analysis:

| **Search Type** | **Best Case** | **Average/Worst Case** |  |
| --- | --- | --- | --- |
| Linear Search | O(1) | O(n) |  |
| Binary Search | O(1) | O(log n) |  |

If the library has few books or frequent updates (where sorting might be costly), linear search is fine.

If the library has many books and search is frequent, binary search is better after sorting the list.