**Week 01 : Superset ID- 6364190**

**ALGORITHM AND DATA STRUCTURE -**

SOLUTIONS:-

Exercise 1:

1. Inventory Management System

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

->As Inventory Management System uses a very large amount of data set which is much complex to use and retrieve . Therefore , data structure helps to handle such situations easily .

The ways data structure helps in this system : -

Frequent Operations - changes frequently in databases like updating items ,removing items and relocating etc .

Fast Retrieval - Millions of data is stored and by using such an algorithm we can retrieve data in seconds .

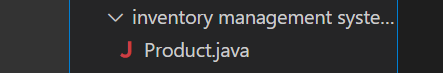
Storing data- Storing data in correct order and algorithm like stack , queue , linked list and array , depending upon their requirements .

Ques) Discuss the types of data structures suitable for this problem.

-> The choice of data structure uses in this model depends on the type of work load .

1. Hash table - To store data frequently hash method works great role.
2. Binary tree - For searching binary tree is best (AVL Tree ).
3. Linked List - To store large data with uneven positions (locations) .
4. Setup:

Create a new project for the inventory management system.



1. Implementation:

CODE:-

import java.util.\*; public class Product

{

private int productId;

private String productName; private int quantity;

private double price;

private static Map<Integer, Product> prodMap = new HashMap<>();

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

{

this.productId = productId; this.productName = productName; this.quantity = quantity;

this.price = price;

}

public void addProduct() {

if (prodMap.containsKey(this.productId))

{

System.out.println("ID already exists");

}

else

{

prodMap.put(this.productId, this); System.out.println("Product added successfully");

}

}

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

{

Product product = prodMap.get(productId); if (product != null) {

product.productName = name; product.quantity = quantity; product.price = price;

System.out.println("updated successfully");

} else {

System.out.println("ID not found.");

}

}

public static void deleteProduct(int productId)

{

if (prodMap.remove(productId) != null) { System.out.println("Product deleted successfully.");

} else {

System.out.println("Product ID not found.");

}

}

public static void displayProducts() { if (prodMap.isEmpty()) {

System.out.println("No products available.");

} else {

for (Product p : prodMap.values()) { System.out.println(p);

}

}

}

public String toString() {

return "ProductID: " + productId + ", Name: " + productName + ", Quantity: " + quantity +

", Price: $" + price;

}

public static void main(String[] args)

{

Product p1 = new Product(51, "LapTop", 10, 850.0); p1.addProduct();

Product p2 = new Product(52, "Phone", 20, 450.0); p2.addProduct();

Product p3 = new Product(53, "TableT", 15, 300.0); p3.addProduct();

System.out.println("\n--- Product List ---"); displayProducts();

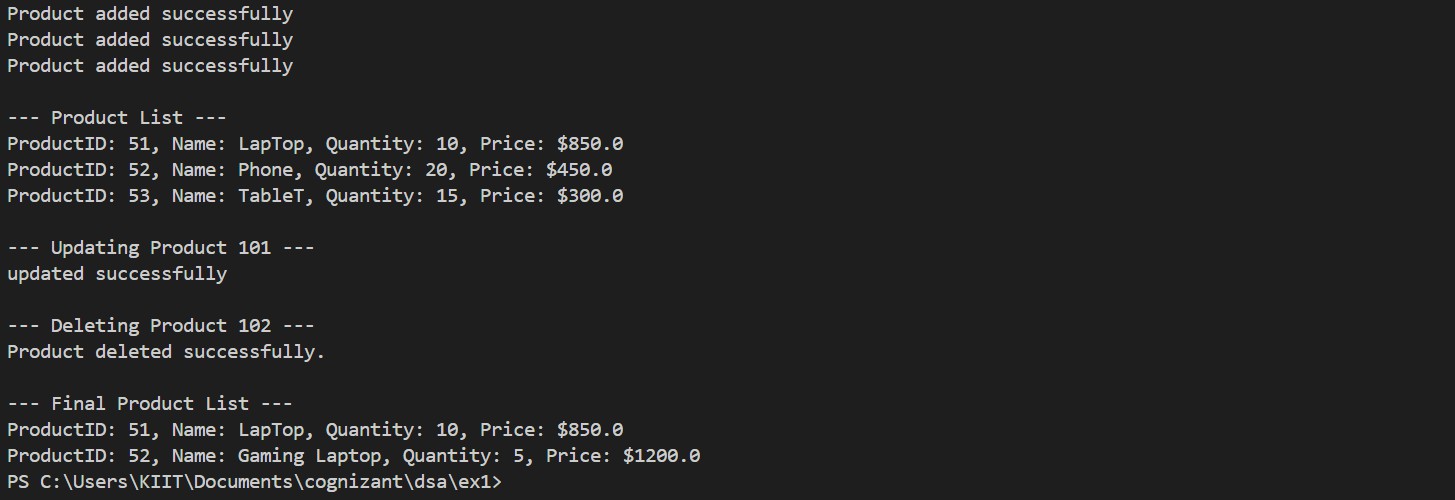
System.out.println("\n--- Updating Product 101 ---"); updateProduct(52, "Gaming Laptop", 5, 1200.0);

System.out.println("\n--- Deleting Product 102 ---"); deleteProduct(53);

System.out.println("\n--- Final Product List ---"); displayProducts();

}

}

output-

1. Analysis : complexity Add operation -
   * Average case -O(1)
   * Worst case - O(n)

update operation -

* + Average Case: O(1)
  + Worst Case: O(n) (hash collisions)

Delete operation -

* + Average case -O(1)
  + Worst case - O(n) (if not found )

Ques)Discuss how you can optimize these operations.

* + -> By using better hash function
  + Minimizing collisions .

# Exercise 2: E-commerce Platform Search Function

Ques) o Explain Big O notation and how it helps in analyzing algorithms.

**-> Big O notation** is a mathematical way to describe how the **runtime or space requirements** of an algorithm grow relative to the **input size .**

It helps in analyzing algorithms by checking their space and time complexity , whether a program will take much time to execute or how much space it's taking .

Ques ) Describe the best, average, and worst-case scenarios for search operations. Linear Search:

* + Best case - O(1)
  + Average case -O(n)
  + Worst case -O(n)

Binary Search:

* + Best case - O(1)
  + Average case -O(log n)
  + Worst case -O(log n)

Hash Table Lookup :

* + Best case - O(1)
  + Average case -O(1)
  + Worst case -O(n)

2.setup 3.implementation CODE:-

import java.util.\*; class Product

{

int productId;

String productName ; String category;

Product(int id,String name,String category)

{

this.productId=id; this.productName=name; this.category=category;

}

static Product prod[]={

new Product(103, "Tablet", "Electronics"), new Product(101, "Laptop", "Electronics"), new Product(105, "Shoes", "Footwear"), new Product(104, "Watch", "Accessories"), new Product(102, "Phone", "Electronics")

};

public static void linearSearch(String name)

{

int c=0; for(Product i:prod)

{

if(i.productName.equalsIgnoreCase(name)) c=1;

}

if(c==1)

System.out.println("Product found : "); else

System.out.println("no product found");

}

public static void binarySearch(int id)

{

Arrays.sort(prod, Comparator.comparingInt(p -> p.productId)); int low = 0, high = prod.length - 1;

while (low <= high)

{

int mid = (low + high) / 2;

if (prod[mid].productId == id)

{

System.out.println(prod[mid]); return ;

} else if (prod[mid].productId < id) { low = mid + 1;

} else {

high = mid - 1;

}

}

}

public static void main(String[] args)

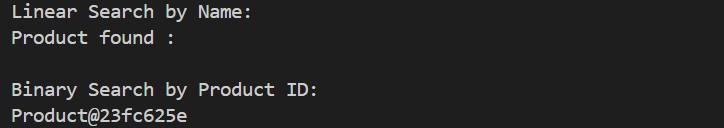
{

System.out.println("Linear Search by Name:"); linearSearch("Phone");

System.out.println("\nBinary Search by Product ID:"); binarySearch(104);

}

}

Output:

Ques)Compare the time complexity of linear and binary search algorithms.

-> Linear Search:

* Best case - O(1)
* Average case -O(n)
* Worst case -O(n)

Binary Search:

* Best case - O(1)
* Average case -O(log n)
* Worst case -O(log n)

Ques)Discuss which algorithm is more suitable for your platform and why.

Binary search ->E-commerce platforms typically deal with **thousands or millions of products**.Binary search offers consistent and predictable performance because it reduces the search space by half with every step.

# Exercise 3: Sorting Customer Orders

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

1. Bubble sort : Compares **adjacent elements** and swaps them if they are in the wrong order.Repeats this process for each element, “bubbling” the largest value to the end in each pass.
   1. Best: O(n) (already sorted)
   2. Average: O(n²)
   3. Worst: O(n²)
2. Insertion Sort: Builds the final sorted array **one element at a time**.Takes each element and **inserts it into its correct position** in the already sorted portion.
   1. Best: O(n) (nearly sorted)
   2. Average: O(n²)
   3. Worst: O(n²)
3. Quick Sort: Uses a **divide-and-conquer** approach.Picks a **pivot** element, then partitions the array into:

* Elements less than pivot
* Elements greater than pivot .Recursively applies the same logic to sub-arrays. a.Best & Average: O(n log n)

b. Worst: O(n²)

4 .Merge Sort: Also uses **divide-and-conquer**. Recursively **divides** the array into halves

. Sorts and merges the halves back into a full sorted array.

1. Best, Average, Worst: O(n log n)
2. Space: O(n)

2and 3 )implementation :

import java.util.\*; public class Order

{

int orderId;

String customerName; double totalPrice;

Order(int id,String name,double price)

{

this.orderId=id; this.customerName=name; this.totalPrice=price;

}

static Order ord[]={

new Order(1001, "shivam", 250.50), new Order(1002, "riya", 180.00), new Order(1003, "parul", 320.75), new Order(1004, "saurabh", 150.25),

new Order(1005, "shubham", 290.10)

};

public static void bubbleSort()

{

int n = ord.length;

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

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

if (ord[j].totalPrice > ord[j + 1].totalPrice)

{

Order temp = ord[j]; ord[j] = ord[j + 1]; ord[j + 1] = temp;

}

}

}

}

public static void quickSort() { quickSort(0, ord.length - 1);

}

// Quick Sort Implementation

private static void quickSort(int low, int high) { if (low < high) {

int pivotIndex = partition(low, high); quickSort(low, pivotIndex - 1); quickSort(pivotIndex + 1, high);

}

}

private static int partition(int low, int high) { double pivot = ord[high].totalPrice;

int i = low - 1;

for (int j = low; j < high; j++) { if (ord[j].totalPrice < pivot) {

i++;

Order temp = ord[i];

ord[i] = ord[j]; ord[j] = temp;

}

}

Order temp = ord[i + 1]; ord[i + 1] = ord[high]; ord[high] = temp;

return i + 1;

}

public static void printord(String title) { System.out.println("\n" + title);

for (Order o : ord) {

System.out.println("OrderID: " + o.orderId + ", Name: " + o.customerName + ", Total:

$" + o.totalPrice);

}

}

public static void main(String[] args) { printord("🔹 Original Order List");

// Bubble Sort bubbleSort();

printord("After Bubble Sort by Total Price");

ord = new Order[]{

new Order(1001, "shivam", 250.50), new Order(1002, "riya", 180.00),

new Order(1003, "parul", 320.75), new Order(1004, "saurabh", 150.25), new Order(1005, "shubham", 290.10)

};

// Quick Sort quickSort();

printord("After Quick Sort by Total Price");

}

}

Output:

**Ques)Compare the performance (time complexity) of Bubble Sort and Quick Sort. Best Case Time O(n) O(n log n)**

**Average Case Time O(n²) O(n log n)**

**Worst Case Time O(n²) O(n²) (rare, poor pivot)**

**Ques) Discuss why Quick Sort is generally preferred over Bubble Sort.**

**Quick Sort** is generally preferred because it is significantly faster for large datasets, uses less memory than Merge Sort, and is often optimized in real-world libraries. **Bubble Sort**, due to its simplicity, is mostly used for educational purposes rather than practical applications.

# Exercise 4: Employee Management System

Ques) Explain how arrays are represented in memory and their advantages.

An **array** is a **contiguous block of memory** that stores elements of the **same data type (similar)**. Each element is stored **next to the previous one** in memory (not random).

Eg: mem location- 10001,10002,10003 ,10004 and so on. Advantages :-

* Direct indexing via memory address calculation.
* Contiguous allocation, no overhead per element.
* Can be looped over using simple loops (for, while, etc.)
* Predictable memory usage, ideal for fixed-size data.

2. Setup and 3. implementation:

CODE :-

import java.util.\*; public class Employee

{

int employeeId ; String name ; String position; double salary;

public Employee(int id,String name ,String position,double salary)

{

this.employeeId=id; this.name=name; this.salary=salary; this.position=position;

}

public static Employee emp[]=new Employee[5]; static int count =0;

static void addEmployee(int id,String name ,String pos,double sal )

{

if(count >= emp.length)

{

System.out.println("employee limit full"); return;

}

emp[count]=new Employee(id, name, pos, sal); count++;

}

public static void searchId(int id)

{

for(Employee i:emp)

{

if(i.employeeId==id)

{System.out.println(i);return;}

}

System.out.println("employee not found ");

}

public static void traverse()

{

if (count == 0)

System.out.println("No employees to display."); for(Employee i:emp)

System.out.println(i);

}

public static void deleteId(int id)

{

int index = -1;

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

if (emp[i].employeeId == id) { index = i;

break;

}

}

if (index == -1)

{

System.out.println("Employee not found."); return;

}

for (int i = index; i < count - 1; i++) { emp[i] = emp[i + 1];

}

emp[--count] = null;

System.out.println("Employee deleted successfully.");

}

public String toString() {

return "ID: " + employeeId + ", Name: " + name + ", Position: " + position + ", Salary: $"

+ salary;

}

public static void main(String[] args)

{

addEmployee(1, "shivam", "Manager", 80000); addEmployee(2, "riya", "Developer", 60000); addEmployee(3, "singhji", "Tester", 50000);

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

traverse();

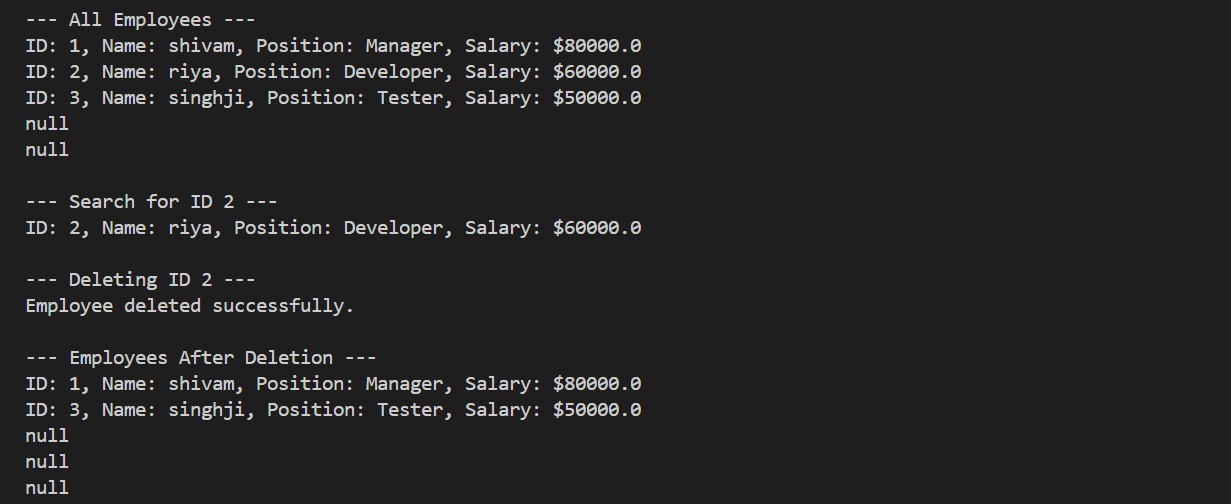
System.out.println("\n--- Search for ID 2 ---"); searchId(2);

System.out.println("\n--- Deleting ID 2 ---"); deleteId(2);

System.out.println("\n--- Employees After Deletion ---"); traverse();

}

}

Output:

# Exercise 5: Task Management System

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

* Singly Linked List

A Singly Linked List is a linear data structure where each node contains two parts: the data and a reference (link) to the next node. The list starts from a special node called the head, and the last node points to null , indicating the end of the list. It allows traversal in only one direction — forward.

Eg : 10->20->30->49->X

* Doubly Linked List

A Doubly Linked List extends the idea of a singly linked list by adding a second pointer — one to the next node, and another to the previous node. Each node contains three parts: data, previous link, and next link. This allows bidirectional traversal — both forward and backward.

Eg : 1⇔2⇔3⇔X

2setup and 3. Implement class node

{

int taskId;

String taskName; String status; node next;

public node(int id, String name, String status) { this.taskId = id;

this.taskName = name; this.status = status; this.next = null;

}

public String toString()

{

return "TaskID: " + taskId + ", Name: " + taskName + ", Status: " + status;

}

}

public class Task

{

static node head = null; static node temp=null;

public static void addTask(int id, String name, String status) {

node newTask = new node(id, name, status);

if (head == null) {

head =temp= newTask;

}

else

{

temp.next=newTask; temp=newTask;

}

System.out.println("Task added: " + name);

}

public static void searchTask(int id) { node current = head;

while (current != null) {

if (current.taskId == id) { System.out.println("Task found: " + current); return;

}

current = current.next;

}

System.out.println("Task with ID " + id + " not found.");

}

public static void traverse() { if (head == null) {

System.out.println("No tasks available."); return;

}

System.out.println("\n--- Task List ---"); node current = head;

while (current != null) { System.out.println(current); current = current.next;

}

}

public static void deleteTask(int id) { if (head == null) {

System.out.println("Task list is empty."); return;

}

if (head.taskId == id) { head = head.next;

System.out.println("Task deleted successfully."); return;

}

node prev = head;

node current = head.next;

while (current != null) {

if (current.taskId == id) { prev.next = current.next;

System.out.println("Task deleted successfully."); return;

}

prev = current; current = current.next;

}

System.out.println("Task with ID " + id + " not found.");

}

public static void main(String[] args) { addTask(1, "Design UI", "Pending");

addTask(2, "Develop Backend", "In Progress"); addTask(3, "Write Tests", "Pending");

traverse();

System.out.println("\n--- Search Task ID 2 ---"); searchTask(2);

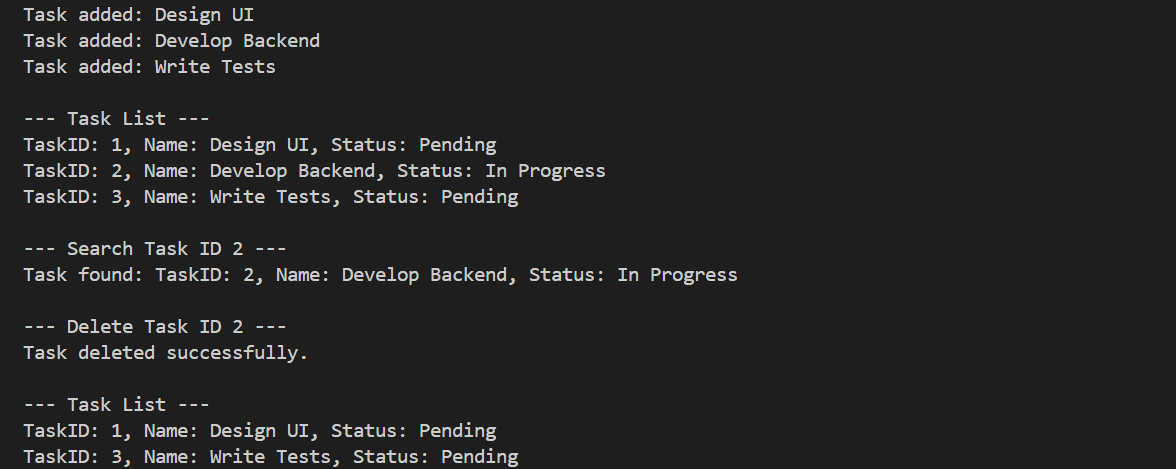
System.out.println("\n--- Delete Task ID 2 ---"); deleteTask(2);

traverse();

}

}

Output:

Ques)Time complexity of each operation :

Add (at end)= O(n) Search (by ID) =O(n) Traverse= O(n) Delete (by ID) =O(n)

Ques) Advantages of Linked Lists Over Arrays

1. Dynamic Size
2. Efficient Insertions/Deletions
3. No Memory Waste

# Exercise 6: Library Management System

Linear Search :

Linear Search is the simplest searching algorithm. It checks **each element one by one** in the array or list until it finds the target value or reaches the end.

Binary Search :

Binary Search is a fast algorithm for **sorted** arrays. It works by repeatedly dividing the search interval in half.

2setup and 3 implement CODE :-

import java.util.Arrays; import java.util.Comparator;

public class Book {

int bookId; String title; String author;

public Book(int id, String title, String author) { this.bookId = id;

this.title = title; this.author = author;

}

static Book[] books = {

new Book(1, "Java ", "James "),

new Book(2, "Python", " Rossum"), new Book(3, "C++", " Stroustrup"), new Book(4, "Dsa", "Robert"),

new Book(5, "Algorithms", "narendra modi")

};

public static Book linearSearch(String title) { for (Book b : books) {

if (b.title.equalsIgnoreCase(title)) { return b;

}

}

return null;

}

public static Book binarySearch(String title) {

Arrays.sort(books, Comparator.comparing(b -> b.title.toLowerCase()));

int low = 0, high = books.length - 1; while (low <= high) {

int mid = (low + high) / 2;

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

if (cmp == 0) return books[mid]; else if (cmp < 0) low = mid + 1; else high = mid - 1;

}

return null;

}

public String toString() {

return "BookID: " + bookId + ", Title: " + title + ", Author: " + author;

}

public static void main(String[] args) { System.out.println("Linear Search: Searching -'Python'"); Book result1 = linearSearch("Python");

System.out.println(result1 != null ? result1 : "Book not found.");

System.out.println("\n Binary Search: Searching -'Dsaa'"); Book result2 = binarySearch("Dsaa");

System.out.println(result2 != null ? result2 : "Book not found.");

}

}

Output :

ques) Compare the time complexity of linear and binary search. Linear search :

* Best -O(1)
* Avg -O(n)
* Worst - O(n) Binary search :
* Best -O(1)
* Avg -O(logn)
* Worst - O(logn)

ques) Discuss when to use each algorithm based on the data set size and order. Linear Search is simple and flexible but slow for large data.

Binary Search is fast and efficient, but it only works on sorted data.

# Exercise 7: Financial Forecasting

**Ques1) Explain the concept of recursion and how it can simplify certain problems.**

**Recursion** is a programming technique where a function **calls itself** to solve smaller instances of the same problem. Each recursive call brings the problem closer to a **base case**, which stops the recursion.

2. setup and 3 implementation :

Code ;

public class FutureValue

{

public static double cal(double presentValue, double rate, int years) { if (years == 0) {

return presentValue;

} else {

return cal(presentValue \* (1 + rate), rate, years - 1);

}

}

public static void main(String[] args) { double presentValue = 1000.0; double growthRate = 0.08;

int years = 5;

double futureValue = cal(presentValue, growthRate, years);

System.out.printf("Future Value after %d years = ₹%.2f\n", years, futureValue);

}

}

Output :

Ques)

Time complexity for recursive algorithm : O(n)

Ques)

How to Optimize the Recursive Solution:

* Use Memoization-Store the results of subproblems in a data structure
* Convert to Iteration- Rewrite the recursive logic as an iterative loop.