**DESIGN PATTERNS AND PRINCIPLES**

**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **SingletonPatternExample**.
2. **Define a Singleton Class:**
   * Create a class named Logger that has a private static instance of itself.
   * Ensure the constructor of Logger is private.
   * Provide a public static method to get the instance of the Logger class.
3. **Implement the Singleton Pattern:**
   * Write code to ensure that the Logger class follows the Singleton design pattern.
4. **Test the Singleton Implementation:**
   * Create a test class to verify that only one instance of Logger is created and used across the application.

CODE :

1. Logger.cs

public class Logger {

*// Singleton class using lazy initialization with synchronization*

    private static Logger? instance;

    private static readonly *object* lockObj *=* *new* *object*();

    private Logger() {

*// Private constructor to prevent instantiation*

    }

    public static Logger GetInstance() {

*if* (instance *==* null) {

*lock* (lockObj) { *//sync for multi threading*

*if* (instance *==* null) {

                    instance *=* *new* Logger();

                }

            }

        }

*return* instance;

    }

*// Logger method*

    public *void* Log(*string* message) {

        Console.WriteLine("[log message]: " *+* message);

    }

}

1. Test.cs

*using* System;

public class Test {

    public static *void* Main(*string*[] args) {

        Logger l1 *=* Logger.GetInstance();

        Logger l2 *=* Logger.GetInstance();

        Console.WriteLine(l1.GetHashCode());

        Console.WriteLine(l2.GetHashCode());

        l1.Log("This is the first log message.");

        l2.Log("This is the second log message.");

*if* (l1.GetHashCode() *==* l2.GetHashCode()) {

            Console.WriteLine("Logger instances are same");

        } *else* {

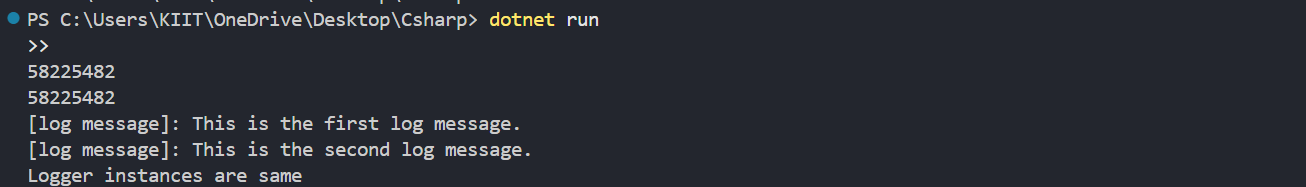
            Console.WriteLine("Logger instances are different");

        }

    }

}

OUTPUT:



**Exercise 2: Implementing the Factory Method Pattern**

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   * Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document Classes:**
   * Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **Implement the Factory Method:**
   * Create an abstract class **DocumentFactory** with a method **createDocument()**.
   * Create concrete factory classes for each document type that extends DocumentFactory and implements the **createDocument()** method.
5. **Test the Factory Method Implementation:**
   * Create a test class to demonstrate the creation of different document types using the factory method.

CODE:

1. IDocument.cs

public interface IDocument

{

*void* Open();

}

1. WordDocument.cs

public class WordDocument : IDocument

{

    public *void* Open()

    {

        Console.WriteLine("Opening a Word document.");

    }

}

1. PDFDocuments.cs

public class WordDocument : IDocument

{

    public *void* Open()

    {

        Console.WriteLine("Opening a Word document.");

    }

}

1. ExcelDocument.cs

public class ExcelDocument : IDocument

{

    public *void* Open()

    {

        Console.WriteLine("Opening an Excel document.");

    }

}

1. DocumentFactory.cs

public abstract class DocumentFactory

{

    public abstract IDocument CreateDocument();

}

public class WordDocumentFactory : DocumentFactory

{

    public override IDocument CreateDocument()

    {

*return* *new* WordDocument();

    }

}

public class PdfDocumentFactory : DocumentFactory

{

    public override IDocument CreateDocument()

    {

*return* *new* PdfDocument();

    }

}

public class ExcelDocumentFactory : DocumentFactory

{

    public override IDocument CreateDocument()

    {

*return* *new* ExcelDocument();

    }

}

1. Program.cs

*using* System;

public class Test

{

    public static *void* Main(*string*[] args)

    {

        DocumentFactory wordFactory *=* *new* WordDocumentFactory();

        IDocument wordDoc *=* wordFactory.CreateDocument();

        wordDoc.Open();

        DocumentFactory pdfFactory *=* *new* PdfDocumentFactory();

        IDocument pdfDoc *=* pdfFactory.CreateDocument();

        pdfDoc.Open();

        DocumentFactory excelFactory *=* *new* ExcelDocumentFactory();

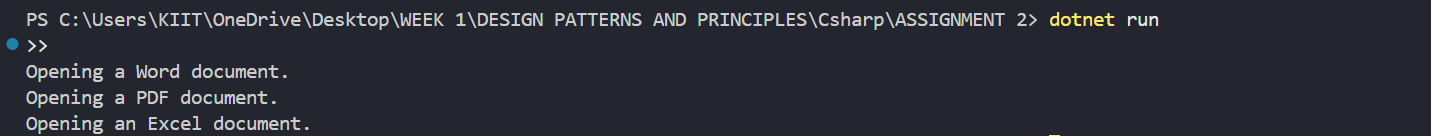
        IDocument excelDoc *=* excelFactory.CreateDocument();

        excelDoc.Open();

    }

}

OUTPUT:



ALGORITHMS DATA STRUCTURES:

**Exercise 1: Inventory Management System**

**Scenario:**

You are developing an inventory management system for a warehouse. Efficient data storage and retrieval are crucial.

**Steps:**

1. **Understand the Problem:**
   * Explain why data structures and algorithms are essential in handling large inventories.
   * Discuss the types of data structures suitable for this problem.
2. **Setup:**
   * Create a new project for the inventory management system.
3. **Implementation:**
   * Define a class Product with attributes like **productId**, **productName**, **quantity**, and **price**.
   * Choose an appropriate data structure to store the products (e.g., ArrayList, HashMap).
   * Implement methods to add, update, and delete products from the inventory.
4. **Analysis:**
   * Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.
   * Discuss how you can optimize these operations.

**Understand the Problem:**

Data structures and algorithms help us to organise a lot of data in an easier way. It prevents degradation of the performance of the code as the program grows and thus helps us to perform the different functions easier.

The two Data structures which can be used here are:

Dictionary (Dictionary<string, Product> ): Ideal for fast lookup by productId, with average O(1) time complexity for insert, update, and delete.

List: Useful if you need ordered traversal, but slower (O(n)) for lookup by ID, so we are not using this here.

CODE:

1. Product.cs

public class Product

{

    public *string* ProductId;

    public *string* ProductName;

    public *int* Quantity;

    public *decimal* Price;

    public Product(*string* id, *string* name, *int* qty, *decimal* price)

    {

        ProductId *=* id;

        ProductName *=* name;

        Quantity *=* qty;

        Price *=* price;

    }

}

1. Inventory.cs

*using* System.Collections.Generic;

public class Inventory

{

    private Dictionary<*string*, Product> products *=* *new* Dictionary<*string*, Product>();

    public *void* AddProduct(Product product)

    {

*if* (*!*products.ContainsKey(product.ProductId))

        {

            products.Add(product.ProductId, product);

            System.Console.WriteLine("Product added.");

        }

*else*

        {

            System.Console.WriteLine("Product ID already exists.");

        }

    }

    public *void* UpdateProduct(Product product)

    {

*if* (products.ContainsKey(product.ProductId))

        {

            products[product.ProductId] *=* product;

            System.Console.WriteLine("Product updated.");

        }

*else*

        {

            System.Console.WriteLine("Product not found.");

        }

    }

    public *void* DeleteProduct(*string* productId)

    {

*if* (products.ContainsKey(productId))

        {

            products.Remove(productId);

            System.Console.WriteLine("Product deleted.");

        }

*else*

        {

            System.Console.WriteLine("Product not found.");

        }

    }

    public *void* ShowProduct(*string* productId)

    {

*if* (products.ContainsKey(productId))

        {

            Product p *=* products[productId];

            System.Console.WriteLine($"ID: {p.ProductId}, Name: {p.ProductName}, Quantity: {p.Quantity}, Price: {p.Price}");

        }

*else*

        {

            System.Console.WriteLine("Product not found.");

        }

    }

}

1. Program.cs

class Program

{

    static *void* Main()

    {

        Inventory inventory *=* *new* Inventory();

        Product p1 *=* *new* Product("101", "Laptop", 10, 55000m);

        Product p2 *=* *new* Product("102", "Mouse", 50, 500m);

        inventory.AddProduct(p1);

        inventory.AddProduct(p2);

        inventory.ShowProduct("101");

        Product updatedP1 *=* *new* Product("101", "Laptop", 15, 55000m);

        inventory.UpdateProduct(updatedP1);

        inventory.ShowProduct("101");

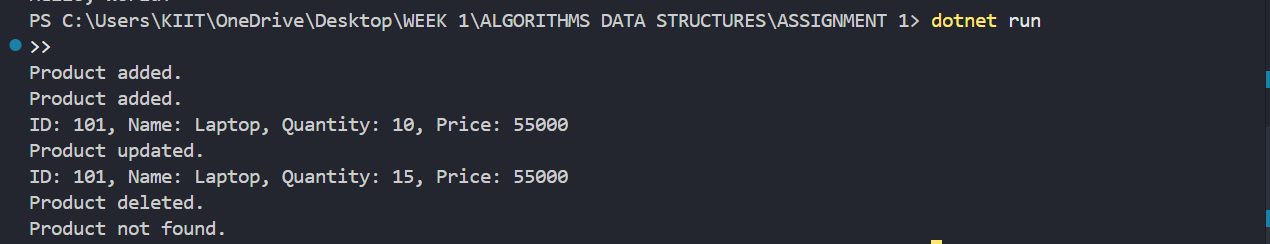
        inventory.DeleteProduct("102");

        inventory.ShowProduct("102");

    }

}

OUTPUT:



ANALYSIS:

For all the functions like add, show, update or delete we need O(1) time complexity as we have used here a HashMap in it. Using the Hashmap we can easily do all the above things very easily and it just takes a O(1) time complexity.

Key:ProductId

Value:Product

For more optimization, we can use a database, which will store large amounts of data and we can also use a cache, which will store frequently used data.

**Exercise 2: E-commerce Platform Search Function**

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

**Understand Asymptotic Notation:**

Big O notation describes how fast or slow an algorithm is as the input size grows.

Constant time – O(1): This means the algorithm takes the same amount of time regardless of the input size. For example, accessing an element in an array by its index is always fast and does not depend on how many elements are in the array.

Linear time – O(n): In this case, the time taken grows directly with the input size. For instance, going through each item in a list to find a value requires time proportional to the number of items, hence linear.

Logarithmic time – O(log n): This is much faster than linear time. Binary search is a good example—each step cuts the input size in half, so the number of steps grows very slowly even as the data grows larger.

Quadratic time – O(n²): These algorithms involve nested loops, meaning for each element, you loop over all other elements. This is common in simple sorting algorithms like bubble sort and becomes inefficient as data size increases.

Search algorithms time complexity:

Linear Search: Best case is O(1) (if the item is first), average and worst cases are O(n) since it may need to check every element.

Binary Search: Best case is O(1) (if the item is in the middle), average and worst cases are O(log n) because it halves the search space each step (requires sorted data).

CODE:

1. Product.cs

public class Product

{

    public *string* ProductId;

    public *string* ProductName;

    public *string* Category;

    public Product(*string* id, *string* name, *string* category)

    {

        ProductId *=* id;

        ProductName *=* name;

        Category *=* category;

    }

}

1. SearchFunctions.cs

*using* System;

public class SearchFunctions

{

    public static *int* LinearSearch(Product[] products, *string* name)

    {

*for* (*int* i *=* 0; i *<* products.Length; i*++*)

        {

*if* (products[i].ProductName *==* name)

            {

*return* i;

            }

        }

*return* *-*1;

    }

    public static *int* BinarySearch(Product[] products, *string* name)

    {

*int* left *=* 0;

*int* right *=* products.Length *-* 1;

*while* (left *<=* right)

        {

*int* mid *=* (left *+* right) */* 2;

*int* result *=* *string*.Compare(products[mid].ProductName, name);

*if* (result *==* 0) *return* mid;

*if* (result *<* 0) left *=* mid *+* 1;

*else* right *=* mid *-* 1;

        }

*return* *-*1;

    }

}

1. Program.cs:

*using* System;

class Program

{

    static *void* Main()

    {

        Product[] products *=* *new* Product[]

        {

*new* Product("101", "Laptop", "Electronics"),

*new* Product("102", "Mouse", "Electronics"),

*new* Product("103", "Shirt", "Clothing"),

*new* Product("104", "Book", "Stationery"),

*new* Product("105", "Pen", "Stationery")

        };

        Console.WriteLine("Linear Search:");

*int* index1 *=* SearchFunctions.LinearSearch(products, "Book");

        Console.WriteLine(index1 *>=* 0 *?* "Found at index " *+* index1 : "Not found");

        Array.Sort(products, (a, b) *=>* a.ProductName.CompareTo(b.ProductName));

        Console.WriteLine("Binary Search:");

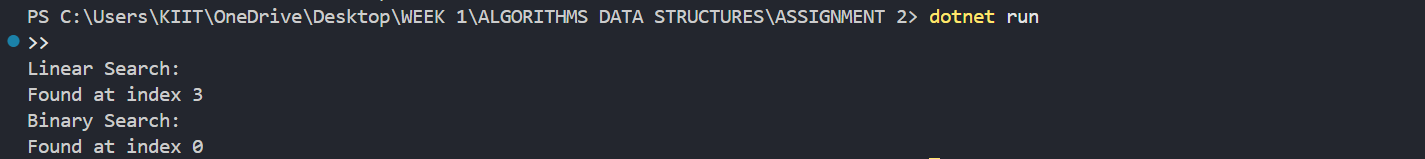
*int* index2 *=* SearchFunctions.BinarySearch(products, "Book");

        Console.WriteLine(index2 *>=* 0 *?* "Found at index " *+* index2 : "Not found");

    }

}

OUTPUT:



ANALYSIS:

Linear Search: O(n) and is used for unsorted lists which are small.

Binary Search: O(log n) and is used for sorted lists which are large.

So for unsorted lists use linear search and for sorted lists use Binary Search.

**Exercise 3: Sorting Customer Orders**

**Scenario:**

You are tasked with sorting customer orders by their total price on an e-commerce platform. This helps in prioritizing high-value orders.

**Steps:**

1. **Understand Sorting Algorithms:**
   * Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).
2. **Setup:**
   * Create a class **Order** with attributes like **orderId**, **customerName**, and **totalPrice**.
3. **Implementation:**
   * Implement **Bubble Sort** to sort orders by **totalPrice**.
   * Implement **Quick Sort** to sort orders by **totalPrice**.
4. **Analysis:**
   * Compare the performance (time complexity) of Bubble Sort and Quick Sort.
   * Discuss why Quick Sort is generally preferred over Bubble Sort.

**Understand Sorting Algorithms:**

Bubble Sort:

Repeatedly compares adjacent elements and swaps them if they're in the wrong order.

Very simple, but inefficient for large data.

Time Complexity: O(n²)

Quick Sort:

Selects a pivot, partitions the array, and recursively sorts subarrays.

Much faster on average for large data sets.

Time Complexity: O(n log n) average, O(n²) worst-case (rare)

CODE:

1. Order.cs

public class Order

{

    public *string* OrderId;

    public *string* CustomerName;

    public *decimal* TotalPrice;

    public Order(*string* id, *string* name, *decimal* price)

    {

        OrderId *=* id;

        CustomerName *=* name;

        TotalPrice *=* price;

    }

}

1. Sorter.cs

*using* System;

public 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].TotalPrice *>* orders[j *+* 1].TotalPrice)

                {

                    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)

    {

*decimal* pivot *=* orders[high].TotalPrice;

*int* i *=* low *-* 1;

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

        {

*if* (orders[j].TotalPrice *<* pivot)

            {

                i*++*;

                Order temp *=* orders[i];

                orders[i] *=* orders[j];

                orders[j] *=* temp;

            }

        }

        Order temp1 *=* orders[i *+* 1];

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

        orders[high] *=* temp1;

*return* i *+* 1;

    }

}

1. Program.cs

*using* System;

class Program

{

    static *void* PrintOrders(Order[] orders)

    {

*foreach* (Order o *in* orders)

        {

            Console.WriteLine($"{o.OrderId} - {o.CustomerName} - ₹{o.TotalPrice}");

        }

    }

    static *void* Main()

    {

        Order[] orders *=* *new* Order[]

        {

*new* Order("O101", "Asmita", 2500),

*new* Order("O102", "Priyanka", 1500),

*new* Order("O103", "Pritha", 3000),

*new* Order("O104", "Aahana", 1200),

*new* Order("O105", "shreshtha", 1800)

        };

        Console.WriteLine("Before Sorting:");

        PrintOrders(orders);

        Console.WriteLine("\nBubble Sort:");

        Sorter.BubbleSort(orders);

        PrintOrders(orders);

        orders *=* *new* Order[]

        {

*new* Order("O101", "Asmita", 2500),

*new* Order("O102", "Priyanka", 1500),

*new* Order("O103", "Pritha", 3000),

*new* Order("O104", "Aahana", 1200),

*new* Order("O105", "shreshtha", 1800)

        };

        Console.WriteLine("\nQuick Sort:");

        Sorter.QuickSort(orders, 0, orders.Length *-* 1);

        PrintOrders(orders);

    }

}

OUTPUT:



ANALYSIS:

Time Complexity:

Bubble Sort

Best Case: O(n) — if the list is already sorted.

Average Case: O(n²) — compares each element with every other.

Worst Case: O(n²) — when the list is in reverse order.

Quick Sort

Best Case: O(n log n) — ideal partitioning with balanced subarrays.

Average Case: O(n log n) — typically efficient in most cases.

Worst Case: O(n²) — if pivot always picks smallest/largest element (rare).

Quick Sort is Preferred as it is much faster on average than Bubble Sort for large datasets.

**Exercise 4: Employee Management System**

**Scenario:**

You are developing an employee management system for a company. Efficiently managing employee records is crucial.

**Steps:**

1. **Understand Array Representation:**
   * Explain how arrays are represented in memory and their advantages.
2. **Setup:**
   * Create a class Employee with attributes like **employeeId**, **name**, **position**, and **salary**.
3. **Implementation:**
   * Use an array to store employee records.
   * Implement methods to **add**, **search**, **traverse**, and **delete** employees in the array.
4. **Analysis:**
   * Analyze the time complexity of each operation (add, search, traverse, delete).
   * Discuss the limitations of arrays and when to use them.

**Understand Array Representation:**

Arrays in memory are contiguous blocks where each element is placed next to the other

The index is used to directly access any element in constant time (O(1)).

The main advantage is faster access.

CODE:

1. Employee.cs

public class Employee

{

    public *int* EmployeeId;

    public *string* Name;

    public *string* Position;

    public *decimal* Salary;

    public Employee(*int* id, *string* name, *string* position, *decimal* salary)

    {

        EmployeeId *=* id;

        Name *=* name;

        Position *=* position;

        Salary *=* salary;

    }

}

1. EmployeeManager.cs

*using* System;

public class EmployeeManager

{

    private Employee[] employees *=* *new* Employee[100];

    private *int* count *=* 0;

    public *void* AddEmployee(Employee emp)

    {

*if* (count *<* employees.Length)

        {

            employees[count] *=* emp;

            count*++*;

            Console.WriteLine("Employee added.");

        }

*else*

        {

            Console.WriteLine("Employee list is full.");

        }

    }

    public *void* SearchEmployee(*int* id)

    {

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

        {

*if* (employees[i].EmployeeId *==* id)

            {

                Console.WriteLine($"{employees[i].EmployeeId} - {employees[i].Name} - {employees[i].Position} - ₹{employees[i].Salary}");

*return*;

            }

        }

        Console.WriteLine("Employee not found.");

    }

    public *void* TraverseEmployees()

    {

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

        {

            Console.WriteLine($"{employees[i].EmployeeId} - {employees[i].Name} - {employees[i].Position} - ₹{employees[i].Salary}");

        }

    }

    public *void* DeleteEmployee(*int* id)

    {

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

        {

*if* (employees[i].EmployeeId *==* id)

            {

*for* (*int* j *=* i; j *<* count *-* 1; j*++*)

                {

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

                }

                employees[count *-* 1] *=* null;

                count*--*;

                Console.WriteLine("Employee deleted.");

*return*;

            }

        }

        Console.WriteLine("Employee not found.");

    }

}

1. Program.cs

class Program

{

    static *void* Main()

    {

        EmployeeManager manager *=* *new* EmployeeManager();

        manager.AddEmployee(*new* Employee(1, "Asmita", "Manager", 80000));

        manager.AddEmployee(*new* Employee(2, "Priyanka", "Developer", 60000));

        manager.AddEmployee(*new* Employee(3, "Pritha", "Designer", 55000));

        Console.WriteLine("\nAll Employees:");

        manager.TraverseEmployees();

        Console.WriteLine("\nSearch Employee with ID 2:");

        manager.SearchEmployee(2);

        Console.WriteLine("\nDelete Employee with ID 2:");

        manager.DeleteEmployee(2);

        Console.WriteLine("\nAll Employees After Deletion:");

        manager.TraverseEmployees();

    }

}

OUTPUT:



ANALYSIS:

Limitations of Arrays:

Fixed Size: Can’t grow beyond the defined limit.

Insert/Delete Costly: Shifting required for mid-array insertions or deletions.

Not Memory-Efficient for sparse or dynamic datasets.

The better alternative is to use lists for better storage and easier use of functions.

**Exercise 5: Task Management System**

**Scenario:**

You are developing a task management system where tasks need to be added, deleted, and traversed efficiently.

**Steps:**

1. **Understand Linked Lists:**
   * Explain the different types of linked lists (Singly Linked List, Doubly Linked List).
2. **Setup:**
   * Create a class **Task** with attributes like **taskId**, **taskName**, and **status**.
3. **Implementation:**
   * Implement a singly linked list to manage tasks.
   * Implement methods to **add**, **search**, **traverse**, and **delete** tasks in the linked list.
4. **Analysis:**
   * Analyze the time complexity of each operation.
   * Discuss the advantages of linked lists over arrays for dynamic data.

**Understand Linked Lists:**

Singly Linked List:

Each node contains data and a next pointer to the next node.

Traverses in one direction only.

Doubly Linked List:

Each node contains data, a next pointer, and a previous pointer.

Allows forward and backward traversal but uses more memory.

CODE:

1. Task.cs

public class Task

{

    public *int* taskId;

    public *string* taskName;

    public *string* status;

    public Task next;

    public Task(*int* id, *string* name, *string* status)

    {

        this.taskId *=* id;

        this.taskName *=* name;

        this.status *=* status;

        this.next *=* null;

    }

}

1. TaskManager.cs

*using* System;

public class TaskManager

{

    private Task head;

    public *void* AddTask(*int* taskId, *string* taskName, *string* status)

    {

        Task newTask *=* *new* Task(taskId, taskName, status);

*if* (head *==* null)

        {

            head *=* newTask;

        }

*else*

        {

            Task current *=* head;

*while* (current.next *!=* null)

            {

                current *=* current.next;

            }

            current.next *=* newTask;

        }

        Console.WriteLine("Task added.");

    }

    public *void* TraverseTasks()

    {

        Task current *=* head;

*while* (current *!=* null)

        {

            Console.WriteLine($"{current.taskId} - {current.taskName} - {current.status}");

            current *=* current.next;

        }

    }

    public *void* SearchTask(*int* taskId)

    {

        Task current *=* head;

*while* (current *!=* null)

        {

*if* (current.taskId *==* taskId)

            {

                Console.WriteLine($"Found: {current.taskId} - {current.taskName} - {current.status}");

*return*;

            }

            current *=* current.next;

        }

        Console.WriteLine("Task not found.");

    }

    public *void* DeleteTask(*int* taskId)

    {

*if* (head *==* null)

        {

            Console.WriteLine("Task list is empty.");

*return*;

        }

*if* (head.taskId *==* taskId)

        {

            head *=* head.next;

            Console.WriteLine("Task deleted.");

*return*;

        }

        Task current *=* head;

*while* (current.next *!=* null)

        {

*if* (current.next.taskId *==* taskId)

            {

                current.next *=* current.next.next;

                Console.WriteLine("Task deleted.");

*return*;

            }

            current *=* current.next;

        }

        Console.WriteLine("Task not found.");

    }

}

1. Program.cs

class Program

{

    static *void* Main()

    {

        TaskManager manager *=* *new* TaskManager();

        manager.AddTask(1, "Design UI", "Pending");

        manager.AddTask(2, "Write Backend", "In Progress");

        manager.AddTask(3, "Testing", "Not Started");

        Console.WriteLine("\nAll Tasks:");

        manager.TraverseTasks();

        Console.WriteLine("\nSearch Task with ID 2:");

        manager.SearchTask(2);

        Console.WriteLine("\nDelete Task with ID 2:");

        manager.DeleteTask(2);

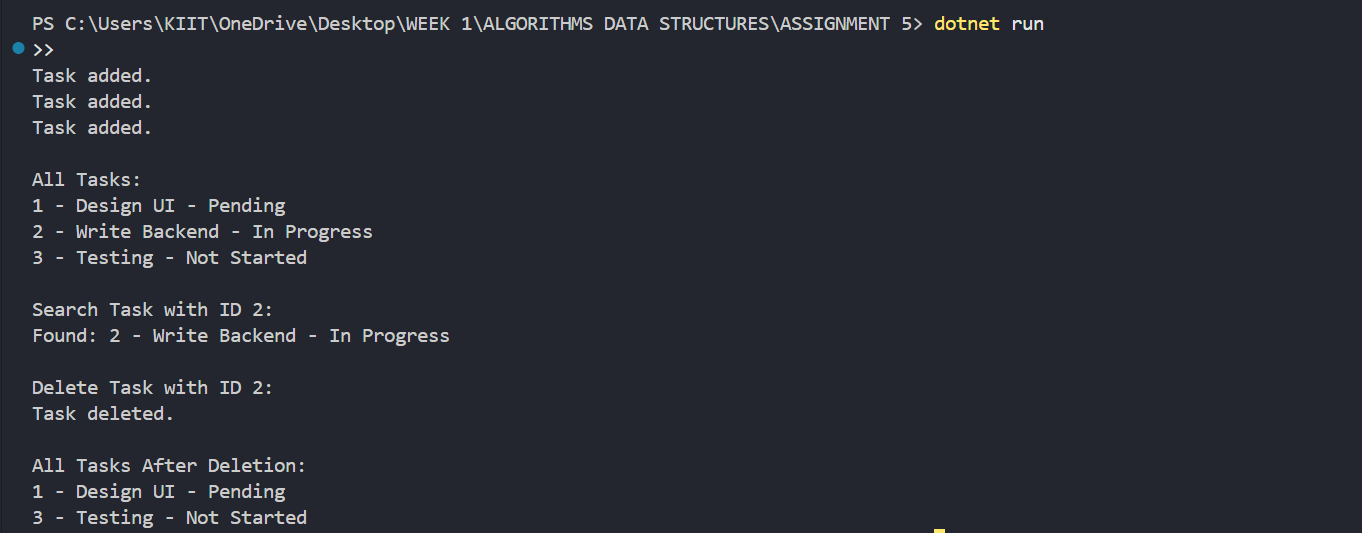
        Console.WriteLine("\nAll Tasks After Deletion:");

        manager.TraverseTasks();

    }

}

OUTPUT:



ANALYSIS:

In a singly linked list, the add, search, traverse, and delete operations all take O(n) time in the worst case because you may need to scan through the entire list to find or place an element. Unlike arrays, there's no direct index-based access, so every operation that involves locating a node requires linear traversal.

Advantages of Linked Lists over Arrays for Dynamic Data:

Linked lists provide better flexibility for dynamic data because they do not require a fixed size like arrays.

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**Understand Recursive Algorithms:**

Recursion is a programming technique where a method calls itself to solve smaller versions of the same problem.

It simplifies certain problem as we can just call the main function rather than constantly using a loop thus reducing the space and time complexity.

CODE:

FutureValue = PresentValue \* (1 + growthRate) ^ years

1. Forecast.cs

*using* System;

class Forecast

{

    public static *double* CalculateFutureValue(*double* presentValue, *double* growthRate, *int* years)

    {

*if* (years *==* 0)

*return* presentValue;

*return* (1 *+* growthRate) *\** CalculateFutureValue(presentValue, growthRate, years *-* 1);

    }

}

1. Program.cs

*using* System;

class Program

{

    static *void* Main()

    {

*double* presentValue *=* 20000;

*double* annualGrowthRate *=* 0.1; *// 10%*

*int* years *=* 5;

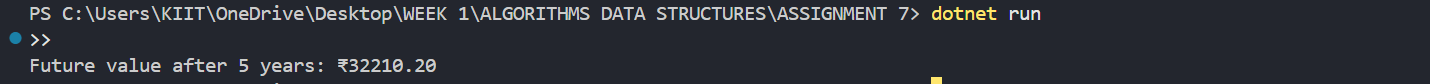
*double* futureValue *=* Forecast.CalculateFutureValue(presentValue, annualGrowthRate, years);

        Console.WriteLine($"Future value after {years} years: ₹{futureValue*:*F2}");

    }

}

OUTPUT:



ANALYSIS:

Time Complexity of Recursive Algorithm:

The recursive function to calculate future value has a time complexity of O(n), where n is the number of years. This is because the function makes one recursive call per year until it reaches the base case (when years == 0).

Optimizing the Recursive Solution:

While this simple recursion works well for small values of n, it can lead to stack overflow or performance issues for large inputs.

To optimize it we can use an iterative approach with a loop instead of recursion.