**Exercise 1: Inventory Management System**

**Filename – Program.cs**

using System;

class Program

{

    static void Main(string[] args)

    {

        InventoryManager manager = new InventoryManager();

        while (true)

        {

            Console.WriteLine("\n Inventory Management Menu ");

            Console.WriteLine("1. Add Product");

            Console.WriteLine("2. Update Product");

            Console.WriteLine("3. Delete Product");

            Console.WriteLine("4. Display Inventory");

            Console.WriteLine("5. Exit");

            Console.Write("Choose an option: ");

            string choice = Console.ReadLine();

            switch (choice)

            {

                case "1":

                    Console.Write("Enter Product ID: ");

                    int idAdd = int.Parse(Console.ReadLine());

                    Console.Write("Enter Product Name: ");

                    string name = Console.ReadLine();

                    Console.Write("Enter Quantity: ");

                    int qtyAdd = int.Parse(Console.ReadLine());

                    Console.Write("Enter Price: ");

                    double priceAdd = double.Parse(Console.ReadLine());

                    manager.AddProduct(new Product(idAdd, name, qtyAdd, priceAdd));

                    break;

                case "2":

                    Console.Write("Enter Product ID to update: ");

                    int idUpdate = int.Parse(Console.ReadLine());

                    Console.Write("Enter New Quantity: ");

                    int qtyUpdate = int.Parse(Console.ReadLine());

                    Console.Write("Enter New Price: ");

                    double priceUpdate = double.Parse(Console.ReadLine());

                    manager.UpdateProduct(idUpdate, qtyUpdate, priceUpdate);

                    break;

                case "3":

                    Console.Write("Enter Product ID to delete: ");

                    int idDelete = int.Parse(Console.ReadLine());

                    manager.DeleteProduct(idDelete);

                    break;

                case "4":

                    manager.DisplayInventory();

                    break;

                case "5":

                    Console.WriteLine("Exiting... Goodbye!");

                    return;

                default:

                    Console.WriteLine("Invalid option. Try again.");

                    break;

            }

        }

    }

}

**Filename – Product.cs**

public class Product

{

    public int ProductId { get; set; }

    public string ProductName { get; set; }

    public int Quantity { get; set; }

    public double Price { get; set; }

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

    {

        ProductId = productId;

        ProductName = productName;

        Quantity = quantity;

        Price = price;

    }

    public override string ToString()

    {

        return $"[{ProductId}] {ProductName} | Qty: {Quantity} | Price: ${Price}";

    }

}

**Filename – InventoryManager.cs**

using System;

using System.Collections.Generic;

public class InventoryManager

{

    private Dictionary<int, Product> inventory = new Dictionary<int, Product>();

    public void AddProduct(Product product)

    {

        inventory[product.ProductId] = product;

    }

    public void UpdateProduct(int productId, int quantity, double price)

    {

        if (inventory.ContainsKey(productId))

        {

            inventory[productId].Quantity = quantity;

            inventory[productId].Price = price;

        }

    }

    public void DeleteProduct(int productId)

    {

        inventory.Remove(productId);

    }

    public void DisplayInventory()

    {

        foreach (var product in inventory.Values)

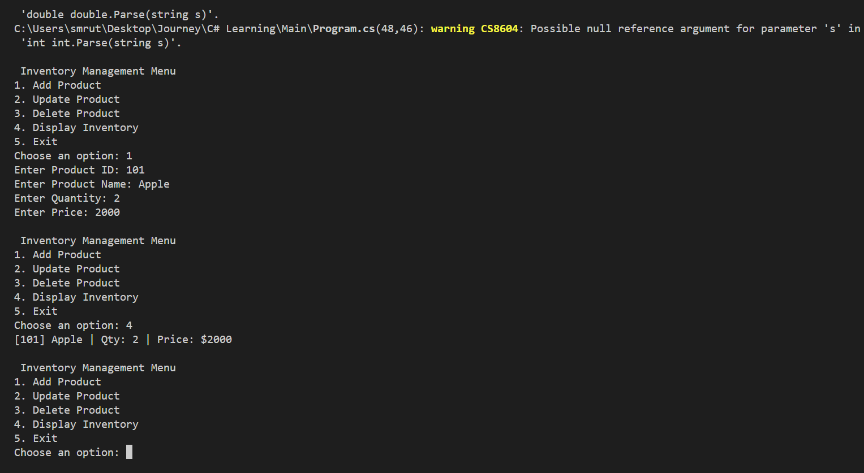
        {

            Console.WriteLine(product);

        }

    }

}

**OUTPUT**

**Analysis --------------------------------------------------------------------------------------------------------------------------**

**For Add: O(1) for HashMap**

**For Update: O(1) for HashMap**

**For Delete: O(1) for HashMap**

**We will use the HashMap here as our main Data Structure for constant-time access via product id.**

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

**Filename – Program.cs**

using System;

public class Product

{

    public int ProductId { get; set; }

    public string ProductName { get; set; }

    public string Category { get; set; }

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

    {

        ProductId = id;

        ProductName = name;

        Category = category;

    }

}

public class SearchFunction

{

    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 cmp = string.Compare(products[mid].ProductName, name);

            if (cmp == 0)

                return mid;

            else if (cmp < 0)

                left = mid + 1;

            else

                right = mid - 1;

        }

        return -1;

    }

}

public class Program

{

    public static void Main()

    {

        Product[] linearProducts = {

            new Product(1, "Laptop", "Electronics"),

            new Product(2, "Shoes", "Footwear"),

            new Product(3, "Watch", "Accessories")

        };

        Product[] binaryProducts = {

            new Product(3, "Laptop", "Electronics"),

            new Product(2, "Shoes", "Footwear"),

            new Product(1, "Watch", "Accessories")

        };

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

        int linearResult = SearchFunction.LinearSearch(linearProducts, "Shoes");

        int binaryResult = SearchFunction.BinarySearch(binaryProducts, "Shoes");

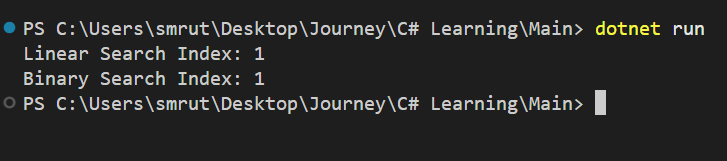
        Console.WriteLine("Linear Search Index: " + linearResult);

        Console.WriteLine("Binary Search Index: " + binaryResult);

    }

}

**OUTPUT**

****

**Analysis --------------------------------------------------------------------------------------------------------------------------**

**Time Complexity:**

**Linear Search: O(n) in all cases**

**Binary Search: O(log n) average/worst, O(1) best (if found in middle)**

**Binary Search is better for large, sorted datasets while Linear Search works slow for large dataset and is for simple and unsorted data.**

**Exercise 3: Sorting Customer Orders**

**Filename – Program.cs**

using System;

public class Order

{

    public int OrderId { get; set; }

    public string CustomerName { get; set; }

    public double TotalPrice { get; set; }

    public Order(int id, string name, double price)

    {

        OrderId = id;

        CustomerName = name;

        TotalPrice = price;

    }

}

public class OrderSorter

{

    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)

                {

                    var 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].TotalPrice;

        int i = low - 1;

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

        {

            if (orders[j].TotalPrice < pivot)

            {

                i++;

                var temp = orders[i];

                orders[i] = orders[j];

                orders[j] = temp;

            }

        }

        var temp1 = orders[i + 1];

        orders[i + 1] = orders[high];

        orders[high] = temp1;

        return i + 1;

    }

}

public class Program

{

    public static void Main()

    {

        Order[] bubbleOrders = {

            new Order(1, "Alice", 500),

            new Order(2, "Bob", 200),

            new Order(3, "Charlie", 800)

        };

        Order[] quickOrders = {

            new Order(1, "Alice", 500),

            new Order(2, "Bob", 200),

            new Order(3, "Charlie", 800)

        };

        OrderSorter.BubbleSort(bubbleOrders);

        OrderSorter.QuickSort(quickOrders, 0, quickOrders.Length - 1);

        Console.WriteLine("Bubble Sort:");

        foreach (var order in bubbleOrders)

            Console.WriteLine(order.OrderId + " " + order.CustomerName + " " + order.TotalPrice);

        Console.WriteLine("Quick Sort:");

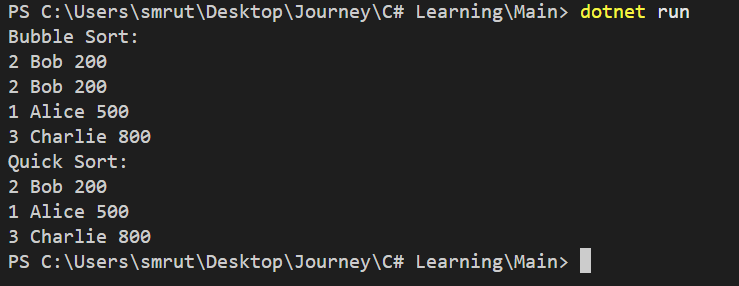
        foreach (var order in quickOrders)

            Console.WriteLine(order.OrderId + " " + order.CustomerName + " " + order.TotalPrice);

    }

}

**OUTPUT**



**Analysis --------------------------------------------------------------------------------------------------------------------------**

**Quick Sort offers O(n log n) average performance while bubble sort is O(n²) and not suitable for large datasets.**

**Exercise 4: Employee Management System**

**Filename – Program.cs**

using System;

public class Employee

{

    public int EmployeeId { get; set; }

    public string Name { get; set; }

    public string Position { get; set; }

    public double Salary { get; set; }

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

    {

        EmployeeId = id;

        Name = name;

        Position = position;

        Salary = salary;

    }

}

public class EmployeeManager

{

    private Employee[] employees;

    private int count;

    public EmployeeManager(int size)

    {

        employees = new Employee[size];

        count = 0;

    }

    public void AddEmployee()

    {

        if (count >= employees.Length)

        {

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

            return;

        }

        Console.Write("Enter ID: ");

        int id = int.Parse(Console.ReadLine());

        Console.Write("Enter Name: ");

        string name = Console.ReadLine();

        Console.Write("Enter Position: ");

        string position = Console.ReadLine();

        Console.Write("Enter Salary: ");

        double salary = double.Parse(Console.ReadLine());

        employees[count] = new Employee(id, name, position, salary);

        count++;

        Console.WriteLine("Employee Added.");

    }

    public void SearchEmployee()

    {

        Console.Write("Enter ID to search: ");

        int id = int.Parse(Console.ReadLine());

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

        {

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

            {

                Console.WriteLine("Found: " + employees[i].Name + " " + employees[i].Position + " " + employees[i].Salary);

                return;

            }

        }

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

    }

    public void Traverse()

    {

        if (count == 0)

        {

            Console.WriteLine("No employees to display.");

            return;

        }

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

        {

            Console.WriteLine(employees[i].EmployeeId + " " + employees[i].Name + " " + employees[i].Position + " " + employees[i].Salary);

        }

    }

    public void DeleteEmployee()

    {

        Console.Write("Enter ID to delete: ");

        int id = int.Parse(Console.ReadLine());

        int index = -1;

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

        {

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

            {

                index = i;

                break;

            }

        }

        if (index == -1)

        {

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

            return;

        }

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

        {

            employees[i] = employees[i + 1];

        }

        employees[count - 1] = null;

        count--;

        Console.WriteLine("Employee Deleted.");

    }

}

public class Program

{

    public static void Main()

    {

        EmployeeManager manager = new EmployeeManager(100);

        while (true)

        {

            Console.WriteLine("\n1. Add Employee");

            Console.WriteLine("2. Search Employee");

            Console.WriteLine("3. Display All Employees");

            Console.WriteLine("4. Delete Employee");

            Console.WriteLine("5. Show Time Complexities");

            Console.WriteLine("6. Exit");

            Console.Write("Choose option: ");

            int choice = int.Parse(Console.ReadLine());

            switch (choice)

            {

                case 1:

                    manager.AddEmployee();

                    break;

                case 2:

                    manager.SearchEmployee();

                    break;

                case 3:

                    manager.Traverse();

                    break;

                case 4:

                    manager.DeleteEmployee();

                    break;

                case 5:

                    Console.WriteLine("Add: O(1)");

                    Console.WriteLine("Search: O(n)");

                    Console.WriteLine("Traverse: O(n)");

                    Console.WriteLine("Delete: O(n)");

                    Console.WriteLine("Limitation: Fixed size, not efficient for dynamic insert/delete.");

                    break;

                case 6:

                    return;

                default:

                    Console.WriteLine("Invalid option.");

                    break;

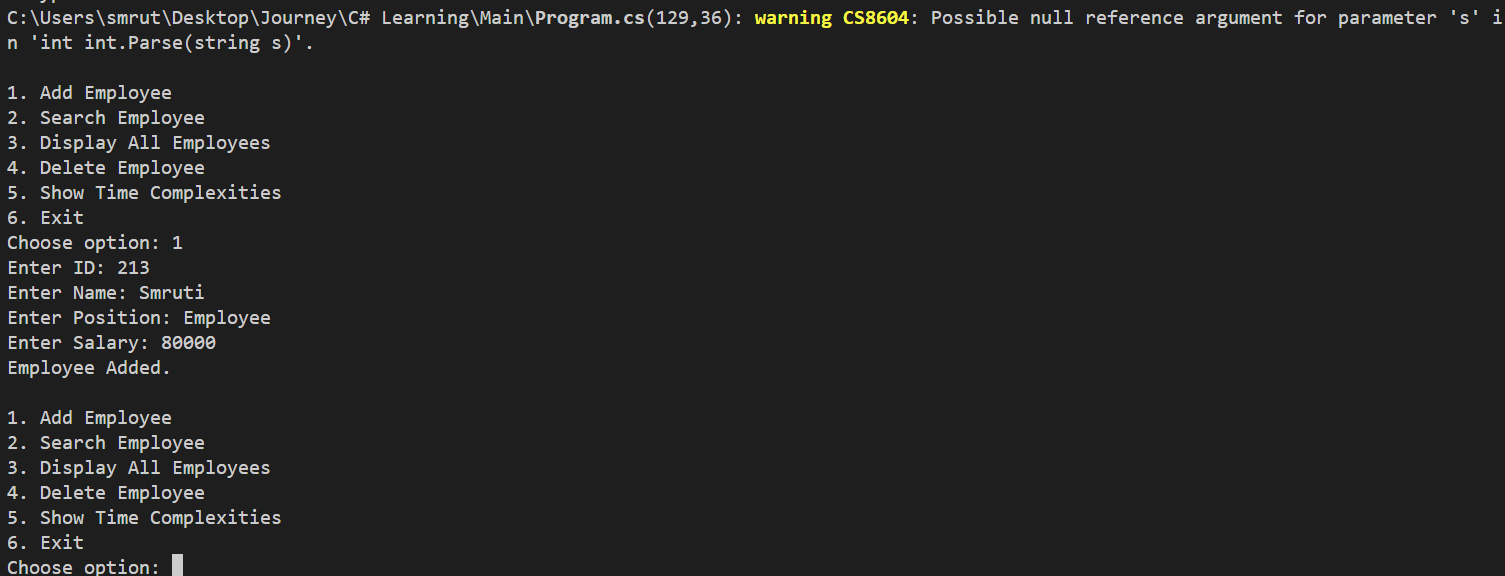
            }

        }

    }

}

**OUTPUT**



**Analysis --------------------------------------------------------------------------------------------------------------------------**

**Arrays allow indexed access and fixed-size record storage. This system uses a stable number of employees but in case the number of employees grow dynamically then a List or Linked List would be appropriate to use.**

**Exercise 5: Task Management System**

**Filename – Program.cs**

using System;

public class Task

{

    public int TaskId { get; set; }

    public string TaskName { get; set; }

    public string Status { get; set; }

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

    {

        TaskId = id;

        TaskName = name;

        Status = status;

    }

}

public class Node

{

    public Task Data;

    public Node Next;

    public Node(Task task)

    {

        Data = task;

        Next = null;

    }

}

public class TaskManager

{

    private Node head;

    public void AddTask()

    {

        Console.Write("Enter Task ID: ");

        int id = int.Parse(Console.ReadLine());

        Console.Write("Enter Task Name: ");

        string name = Console.ReadLine();

        Console.Write("Enter Task Status: ");

        string status = Console.ReadLine();

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

        Node newNode = new Node(newTask);

        if (head == null)

            head = newNode;

        else

        {

            Node temp = head;

            while (temp.Next != null)

                temp = temp.Next;

            temp.Next = newNode;

        }

        Console.WriteLine("Task Added.");

    }

    public void SearchTask()

    {

        Console.Write("Enter Task ID to search: ");

        int id = int.Parse(Console.ReadLine());

        Node temp = head;

        while (temp != null)

        {

            if (temp.Data.TaskId == id)

            {

                Console.WriteLine("Found: " + temp.Data.TaskName + " - " + temp.Data.Status);

                return;

            }

            temp = temp.Next;

        }

        Console.WriteLine("Task Not Found.");

    }

    public void TraverseTasks()

    {

        Node temp = head;

        if (temp == null)

        {

            Console.WriteLine("No Tasks Available.");

            return;

        }

        while (temp != null)

        {

            Console.WriteLine(temp.Data.TaskId + " " + temp.Data.TaskName + " " + temp.Data.Status);

            temp = temp.Next;

        }

    }

    public void DeleteTask()

    {

        Console.Write("Enter Task ID to delete: ");

        int id = int.Parse(Console.ReadLine());

        if (head == null)

        {

            Console.WriteLine("No Tasks to Delete.");

            return;

        }

        if (head.Data.TaskId == id)

        {

            head = head.Next;

            Console.WriteLine("Task Deleted.");

            return;

        }

        Node prev = head;

        Node curr = head.Next;

        while (curr != null)

        {

            if (curr.Data.TaskId == id)

            {

                prev.Next = curr.Next;

                Console.WriteLine("Task Deleted.");

                return;

            }

            prev = curr;

            curr = curr.Next;

        }

        Console.WriteLine("Task Not Found.");

    }

    public void ShowTimeComplexities()

    {

        Console.WriteLine("Add: O(n)");

        Console.WriteLine("Search: O(n)");

        Console.WriteLine("Traverse: O(n)");

        Console.WriteLine("Delete: O(n)");

        Console.WriteLine("Linked List Advantage: Dynamic size, efficient insert/delete without shifting.");

    }

}

public class Program

{

    public static void Main()

    {

        TaskManager manager = new TaskManager();

        while (true)

        {

            Console.WriteLine("\n1. Add Task");

            Console.WriteLine("2. Search Task");

            Console.WriteLine("3. Traverse Tasks");

            Console.WriteLine("4. Delete Task");

            Console.WriteLine("5. Show Time Complexities");

            Console.WriteLine("6. Exit");

            Console.Write("Choose Option: ");

            int choice = int.Parse(Console.ReadLine());

            switch (choice)

            {

                case 1:

                    manager.AddTask();

                    break;

                case 2:

                    manager.SearchTask();

                    break;

                case 3:

                    manager.TraverseTasks();

                    break;

                case 4:

                    manager.DeleteTask();

                    break;

                case 5:

                    manager.ShowTimeComplexities();

                    break;

                case 6:

                    return;

                default:

                    Console.WriteLine("Invalid Option.");

                    break;

            }

        }

    }

}

**OUTPUT**

**Analysis --------------------------------------------------------------------------------------------------------------------------**

**In this case , singly linked list is used as it allows efficient insertions and deletions without shifting elements.**

**Tasks are added, removed, and updated frequently , linked list supports dynamic sizing and avoids performance penalties as seen with arrays during deletion or insertion.**

**Exercise 6: Library Management System**

**Filename – Program.cs**

using System;

public class Book

{

    public int BookId { get; set; }

    public string Title { get; set; }

    public string Author { get; set; }

    public Book(int id, string title, string author)

    {

        BookId = id;

        Title = title;

        Author = author;

    }

}

public class Library

{

    private Book[] books;

    private int count;

    public Library(int size)

    {

        books = new Book[size];

        count = 0;

    }

    public void AddBook()

    {

        if (count >= books.Length)

        {

            Console.WriteLine("Library full.");

            return;

        }

        Console.Write("Enter Book ID: ");

        int id = int.Parse(Console.ReadLine());

        Console.Write("Enter Title: ");

        string title = Console.ReadLine();

        Console.Write("Enter Author: ");

        string author = Console.ReadLine();

        books[count] = new Book(id, title, author);

        count++;

        Console.WriteLine("Book Added.");

    }

    public void LinearSearchByTitle()

    {

        Console.Write("Enter title to search: ");

        string searchTitle = Console.ReadLine();

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

        {

            if (books[i].Title.Equals(searchTitle, StringComparison.OrdinalIgnoreCase))

            {

                Console.WriteLine("Found: " + books[i].BookId + " " + books[i].Title + " by " + books[i].Author);

                return;

            }

        }

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

    }

    public void BinarySearchByTitle()

    {

        Console.Write("Enter title to search: ");

        string searchTitle = Console.ReadLine();

        Array.Sort(books, 0, count, Comparer<Book>.Create((a, b) => a.Title.CompareTo(b.Title)));

        int left = 0;

        int right = count - 1;

        while (left <= right)

        {

            int mid = (left + right) / 2;

            int cmp = string.Compare(books[mid].Title, searchTitle, true);

            if (cmp == 0)

            {

                Console.WriteLine("Found: " + books[mid].BookId + " " + books[mid].Title + " by " + books[mid].Author);

                return;

            }

            else if (cmp < 0)

                left = mid + 1;

            else

                right = mid - 1;

        }

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

    }

    public void DisplayBooks()

    {

        if (count == 0)

        {

            Console.WriteLine("No books available.");

            return;

        }

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

        {

            Console.WriteLine(books[i].BookId + " " + books[i].Title + " by " + books[i].Author);

        }

    }

    public void ShowTimeComplexity()

    {

        Console.WriteLine("Linear Search: O(n)");

        Console.WriteLine("Binary Search: O(log n) (only if sorted)");

        Console.WriteLine("Use Linear Search for small/unsorted datasets.");

        Console.WriteLine("Use Binary Search for large/sorted datasets.");

    }

}

public class Program

{

    public static void Main()

    {

        Library library = new Library(100);

        while (true)

        {

            Console.WriteLine("\n1. Add Book");

            Console.WriteLine("2. Display Books");

            Console.WriteLine("3. Linear Search by Title");

            Console.WriteLine("4. Binary Search by Title");

            Console.WriteLine("5. Show Time Complexity Info");

            Console.WriteLine("6. Exit");

            Console.Write("Choose Option: ");

            int choice = int.Parse(Console.ReadLine());

            switch (choice)

            {

                case 1:

                    library.AddBook();

                    break;

                case 2:

                    library.DisplayBooks();

                    break;

                case 3:

                    library.LinearSearchByTitle();

                    break;

                case 4:

                    library.BinarySearchByTitle();

                    break;

                case 5:

                    library.ShowTimeComplexity();

                    break;

                case 6:

                    return;

                default:

                    Console.WriteLine("Invalid Option.");

                    break;

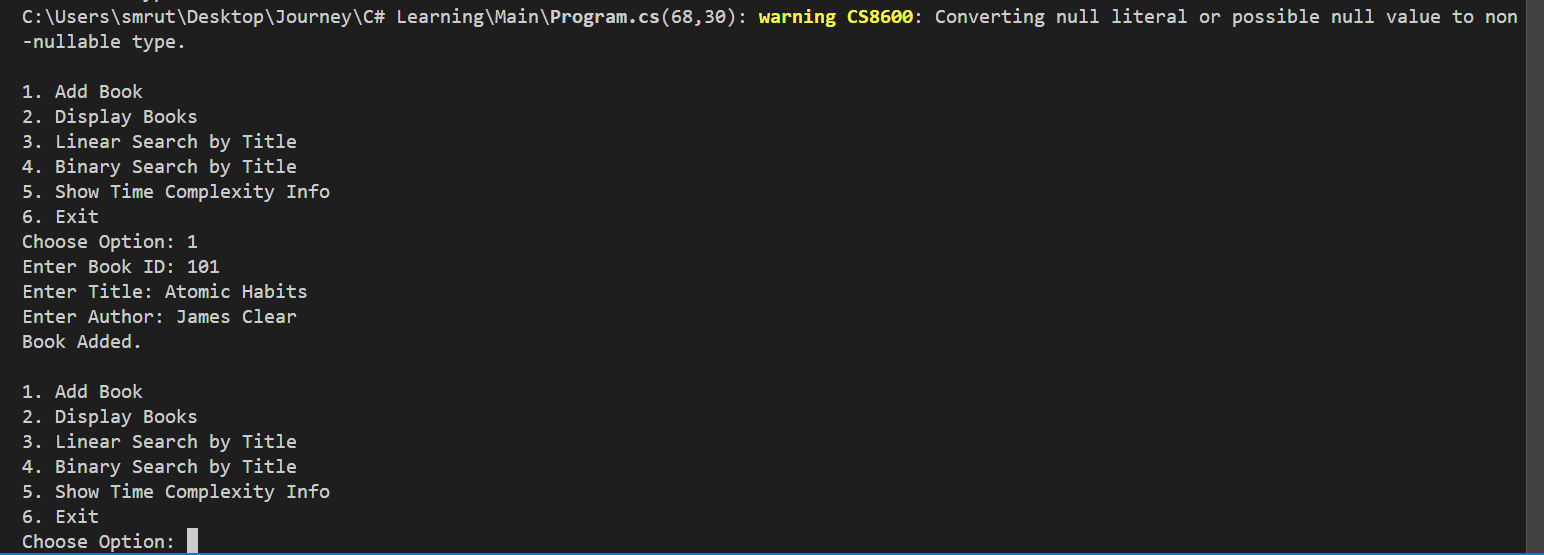
            }

        }

    }

}

**OUTPUT**



**Analysis --------------------------------------------------------------------------------------------------------------------------**

**Binary Search has better speed when data is sorted and in this case searching by title is the main operation.**

**For libraries with large collections, maintaining a sorted array of books enables fast binary search. For small/unsorted libraries, linear search is easier and flexible.**

**Data Structure used is array.**

**Exercise 7: Financial Forecasting**

**Filename – Program.cs**

using System;

public class FinancialForecast

{

    public static double PredictFutureValue(double initialValue, double growthRate, int years)

    {

        if (years == 0)

            return initialValue;

        return PredictFutureValue(initialValue, growthRate, years - 1) \* (1 + growthRate);

    }

    public static double PredictFutureValueMemo(double initialValue, double growthRate, int years, double[] memo)

    {

        if (years == 0)

            return initialValue;

        if (memo[years] != 0)

            return memo[years];

        memo[years] = PredictFutureValueMemo(initialValue, growthRate, years - 1, memo) \* (1 + growthRate);

        return memo[years];

    }

}

public class Program

{

    public static void Main()

    {

        Console.Write("Enter Initial Value: ");

        double initialValue = double.Parse(Console.ReadLine());

        Console.Write("Enter Annual Growth Rate (e.g., 0.05 for 5%): ");

        double growthRate = double.Parse(Console.ReadLine());

        Console.Write("Enter Number of Years: ");

        int years = int.Parse(Console.ReadLine());

        double result = FinancialForecast.PredictFutureValue(initialValue, growthRate, years);

        Console.WriteLine("Future Value (Recursive): " + result);

        double[] memo = new double[years + 1];

        double resultMemo = FinancialForecast.PredictFutureValueMemo(initialValue, growthRate, years, memo);

        Console.WriteLine("Future Value (Memoized): " + resultMemo);

        Console.WriteLine("\nTime Complexity:");

        Console.WriteLine("Recursive: O(n)");

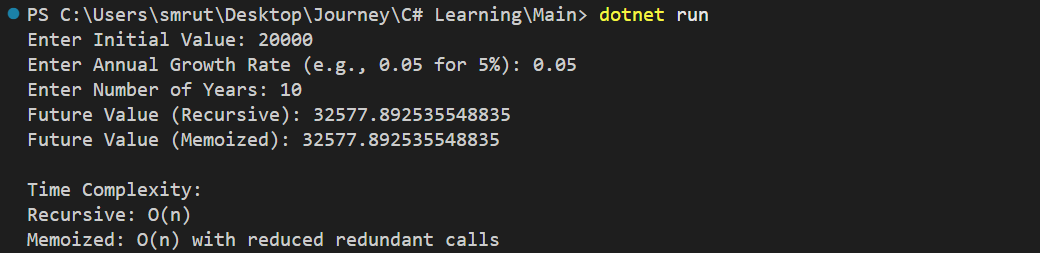
        Console.WriteLine("Memoized: O(n) with reduced redundant calls");

        Console.WriteLine("Optimization: Use memoization to cache intermediate results and avoid recomputation.");

    }

}

**OUTPUT**

****

**Analysis --------------------------------------------------------------------------------------------------------------------------**

**Recursion simplifies modeling future values based on past trends while memoization avoids recomputation and financial forecasts often depend on repeated calculations over previous years. Hence a recursive model with memoization ensures clarity and performance.**