**Exercise 1: Inventory Management System**

1. Data structures and algorithms are essential in handling large inventories because-

* **Efficiency:** Warehouses manage thousands (or more) of products. Efficient search, insert, update, and delete operations are key to fast inventory lookup and modification.
* **Scalability:** Good algorithms ensure performance remains stable even as data grows.
* **Optimization:** Reduces time and space complexity; e.g., HashMap provides constant time lookups vs. ArrayList which takes linear time.
* **Data Organization:** Appropriate data structures like trees, lists, maps allow us to organize and access data effectively based on different needs (ID-based lookup, sorting, etc.).

Data Structures sutiabe for this problem are-

* **ArrayList:** When order matters and data is small. Inefficient in insertiona nd deletion and searchinf when data is very large.
* **HashMap:** When fast lookups are essential as they provide consant lookup times. Unordered key must be unique.
* **LinkedList:** Provides efficient delete and insert at ends but provides poorrandom access speed.

The best suited data structure for thisproblem would be a **HashMap** as it porivdes constant lookup times and products can be searched by their **ProductId.**

**Implementation:**

using System.IO.Pipes;

using System.Xml.Serialization;

public class Product

{

    public string productId;

    public String productName;

    public int quantity;

    public int price;

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

    {

        this.productId = productId;

        this.productName = productName;

        this.quantity = quantity;

        this.price = price;

    }

}

class Program

{

    public static Dictionary<string, Product> productMap = new Dictionary<String, Product>();

    public void AddProduct()

    {

        Console.WriteLine("Enter productId");

        string id = Console.ReadLine();

        Console.WriteLine("Enter product name");

        string name = Console.ReadLine();

        Console.WriteLine("Enter quantity");

        int quantity = Convert.ToInt32(Console.ReadLine());

        Console.WriteLine("Enter Price");

        int price = Convert.ToInt32(Console.ReadLine());

        productMap.Add(id, new Product(id, name, quantity, price));

        Console.WriteLine("Product Entered");

    }

    public void UpdateProduct()

    {

        Console.WriteLine("Enter productId to update");

        string updateId = Console.ReadLine();

        if (productMap.ContainsKey(updateId))

        {

            Console.WriteLine("what product paramete wold you like to update");

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

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

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

            int ch = Convert.ToInt32(Console.ReadLine());

            switch (ch)

            {

                case 1:

                    Console.WriteLine("Enter new name");

                    productMap[updateId].productName = Console.ReadLine();

                    break;

                case 2:

                    Console.WriteLine("Enter new product quantity");

                    productMap[updateId].quantity = Convert.ToInt32(Console.ReadLine());

                    break;

                case 3:

                    Console.WriteLine("Enter new product price");

                    productMap[updateId].price = Convert.ToInt32(Console.ReadLine());

                    break;

                default:

                    Console.WriteLine("Wronf choice entered");

                    break;

            }

        }

        else

        {

            Console.WriteLine("product does not exist in the map");

        }

    }

    public void DeleteProduct()

    {

        Console.WriteLine("Enter product id to delete");

        string id = Console.ReadLine();

        if (productMap.Count() != 0)

        {

            if (productMap.ContainsKey(id))

            {

                productMap.Remove(id);

            }

            else

            {

                Console.WriteLine("product doesnot exist");

            }

        }

        else

        {

            Console.WriteLine("No products to remove");

        }

    }

    public static void Main(String[] args)

    {

        Program obj = new Program();

        while (true)

        {

            Console.WriteLine("Choose as option--->>>");

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

            Console.WriteLine("2. Update a product");

            Console.WriteLine("3. Delete a product");

            Console.WriteLine("4. To exit");

            int choice = Convert.ToInt32(Console.ReadLine());

            switch (choice)

            {

                case 1:

                    obj.AddProduct();

                    break;

                case 2:

                    obj.UpdateProduct();

                    break;

                case 3:

                    obj.DeleteProduct();

                    break;

                case 4:

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

                    return;

                default:

                    Console.WriteLine("Wrong choice");

                    break;

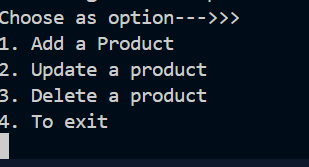
            }

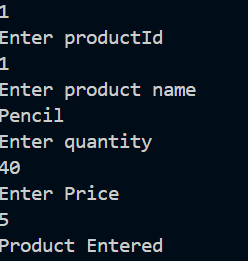
        }

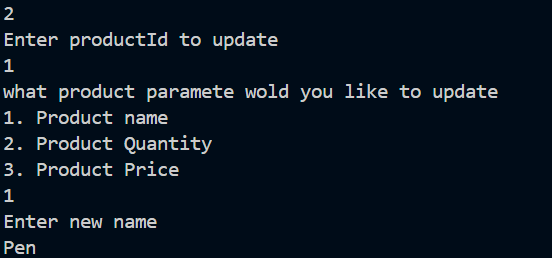
    }

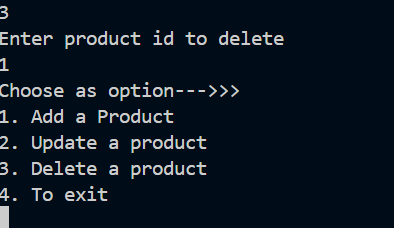
}

**OUTPUT:**









1. **Analysis**

**Time complexity**

* Adding a product - O(1) Average
* Deleting a Product - O(1) Average
* Updating a product - O(1) Average

**Optimization Ideas:**

* **Indexing by other fields:** Use a second Dictionary<string, List<Product>> for productName-based lookup.
* **Concurrent Access:** Use ConcurrentDictionary if multiple threads might access inventory.
* **Persistent Storage:** Use a database (SQL or NoSQL) when data size is too large for memory.
* **Sorting Support:** Use SortedDictionary or sort the values when needed.

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

1)Big O notation is used to describe the time or space complexity of an algorithm in the worst-case scenario as the input size (n) grows. It helps us compare algorithms based on how well they scale.

For example:

* **O(1)** – Constant time (fastest)
* **O(log n)** – Logarithmic time (efficient)
* **O(n)** – Linear time
* **O(n log n)** – Log-linear (e.g., efficient sorting)
* **O(n^2)** – Quadratic (slow for large n)

**Best Average and worst case of sorting alogrithm :-**

|  |  |  |  |
| --- | --- | --- | --- |
| Searching Algorithm | Best Case | Average Case | Worst Case |
| Linear Search | O(1) (first match) | O(n/2) => O(n) (element is somewhere in the middle) | O(n) (Element is at the end of the list) |
| Binary Search | O(1) (element is present at the middle of the list) | O(log n) | O(log n) |

**Implementation**

using System;

class Product

{

    public int ProductId;

    public string ProductName;

    public string Category;

    public Product(int ProductId, string ProductName, string Category)

    {

        this.ProductId = ProductId;

        this.ProductName = ProductName;

        this.Category = Category;

    }

}

class Program

{

    public void LinearSearch(Product[] products)

    {

        Console.WriteLine("Enter product ID to search:");

        int key = Convert.ToInt32(Console.ReadLine());

        foreach (var product in products)

        {

            if (product.ProductId == key)

            {

                Console.WriteLine("Product found");

                return;

            }

        }

        Console.WriteLine("Product not found");

    }

    public void BinarySearch(Product[] products)

    {

        Console.WriteLine("Enter product ID to search:");

        int key = Convert.ToInt32(Console.ReadLine());

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

        int l = 0;

        int r = products.Length - 1;

        while (l <= r)

        {

            int mid = (l + r) / 2;

            if (products[mid].ProductId == key)

            {

                Console.WriteLine($"Product found at index {mid}");

                return;

            }

            else if (products[mid].ProductId < key)

            {

                l = mid + 1;

            }

            else

            {

                r = mid - 1;

            }

        }

        Console.WriteLine("Product not found");

    }

    public static void Main(string[] args)

    {

        Product[] products = new Product[]

        {

            new Product(15, "Smartwatch", "Electronics"),

            new Product(1, "Shampoo", "Personal Care"),

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

            new Product(2, "Phone", "Electronics"),

            new Product(4, "Tablet", "Electronics"),

            new Product(10, "Headphones", "Electronics"),

            new Product(8, "Charger", "Accessories"),

            new Product(7, "Keyboard", "Accessories"),

            new Product(6, "Mouse", "Accessories"),

            new Product(9, "Monitor", "Electronics"),

            new Product(12, "Webcam", "Electronics"),

            new Product(11, "Speaker", "Electronics"),

            new Product(5, "Notebook", "Stationery"),

            new Product(14, "Power Bank", "Accessories"),

            new Product(13, "External Hard Drive", "Storage"),

            new Product(16, "Graphics Card", "Components"),

            new Product(17, "SSD Drive", "Storage"),

            new Product(18, "Bluetooth Adapter", "Accessories"),

            new Product(19, "Router", "Networking"),

            new Product(20, "Microphone", "Audio")

        };

        Program obj = new Program();

        Console.WriteLine("Choose searching algorithm:");

        Console.WriteLine("1. Linear search");

        Console.WriteLine("2. Binary search");

        int choice = Convert.ToInt32(Console.ReadLine());

        switch (choice)

        {

            case 1:

                obj.LinearSearch(products);

                break;

            case 2:

                obj.BinarySearch(products);

                break;

            default:

                Console.WriteLine("Wrong choice");

                break;

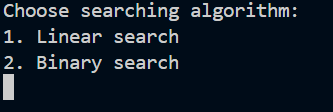
        }

    }

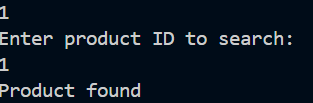
}

**OUTPUT**

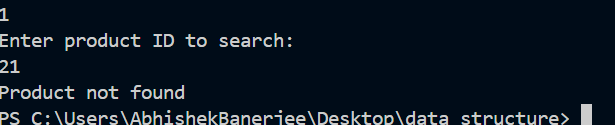
Menu



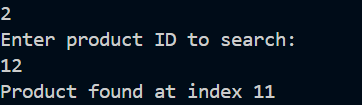
Linear Search with successful search



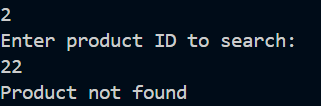
Linear Search with unsuccessful search



Binary Search with successful search results



Binary Search with unsuccessful result



|  |  |  |  |
| --- | --- | --- | --- |
| Searching Algorithm | Best Case | Average Case | Worst Case |
| Linear Search | O(1) (first match) | O(n/2) => O(n) (element is somewhere in the middle) | O(n) (Element is at the end of the list) |
| Binary Search | O(1) (element is present at the middle of the list) | O(log n) | O(log n) |

The best suited algorithm for this problem would be Binary search as it provides fast search time but the list needs to be sorted in order for it to work.

**Exercise 3: Sorting Customer Orders**

1)

**1. Bubble Sort**

How it works:

* Repeatedly steps through the list.
* Compares adjacent elements and swaps them if they are in the wrong order.
* Largest element "bubbles up" to the end after each pass.

Example:

[5, 3, 8, 4] → [3, 5, 4, 8] → [3, 4, 5, 8]

Time Complexity:

**Best:** O(n) (already sorted)

**Average/Worst:** O(n^2)

**Space:** O(1)

**Use Case:** Educational; not suitable for large data.

2. **Insertion Sort**

* How it works:
* Builds the sorted array one item at a time.
* Takes each element and inserts it into its correct position in the sorted part.

**Example:**

[5, 3, 8, 4] → [3, 5, 8, 4] → [3, 4, 5, 8]

**Time Complexity:**

**Best:** O(n) (already sorted)

**Average/Worst:** O(n^2)

**Space:** O(1)

Use Case: Small datasets, nearly sorted data.

3. **Quick Sort**

How it works:

* Picks a pivot.
* Partitions the array so that:
* Elements < pivot on the left
* Elements > pivot on the right
* Recursively sorts left and right partitions.

Example (Pivot = 5):

[5, 3, 8, 4] → [3, 4] [5] [8]

Time Complexity:

**Best/Average:** O(n log n)

**Worst:** O(n^2) (e.g., sorted array with bad pivot)

**Space:** O(log n) (stack)

Use Case: Fast, widely used for large datasets.

4. **Merge Sort**

How it works:

* Divide the list into halves recursively.
* Conquer by sorting the halves.
* Combine by merging sorted halves.

**Example:**

[5, 3, 8, 4] → [5,3], [8,4] → [3,5], [4,8] → [3,4,5,8]

**Time Complexity:**

**Best/Average/Worst:** O(n log n)

**Space:** O(n)

**Use Case:** Large datasets, stable sort, parallel systems.

**Implementation**

using System;

class Order

{

    public int OrderId;

    public string CustomerName;

    public double TotalPrice;

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

    {

        OrderId = id;

        CustomerName = name;

        TotalPrice = price;

    }

    public void Display()

    {

        Console.WriteLine($"Order ID: {OrderId}, Customer: {CustomerName}, Total Price: {TotalPrice}");

    }

}

class Program

{

    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 temp2 = orders[i + 1];

        orders[i + 1] = orders[high];

        orders[high] = temp2;

        return i + 1;

    }

    public static void DisplayOrders(Order[] orders)

    {

        foreach (var order in orders)

        {

            order.Display();

        }

    }

    static void Main()

    {

        Order[] orders = {

            new Order(101, "Alice", 2500.50),

            new Order(102, "Bob", 1800.75),

            new Order(103, "Charlie", 3200.00),

            new Order(104, "David", 900.00),

            new Order(105, "Eve", 2700.30)

        };

        while (true)

        {

            Console.WriteLine("\n Sort Customer Orders");

            Console.WriteLine("1. Display Orders");

            Console.WriteLine("2. Sort using Bubble Sort");

            Console.WriteLine("3. Sort using Quick Sort");

            Console.WriteLine("4. Exit");

            Console.Write("Enter your choice: ");

            if (!int.TryParse(Console.ReadLine(), out int choice))

            {

                Console.WriteLine("Invalid input. Enter a number.");

                continue;

            }

            // Create a fresh copy of orders for sorting

            Order[] ordersCopy = (Order[])orders.Clone();

            switch (choice)

            {

                case 1:

                    Console.WriteLine("\nOriginal Orders:");

                    DisplayOrders(orders);

                    break;

                case 2:

                    Console.WriteLine("\nSorted using Bubble Sort:");

                    BubbleSort(ordersCopy);

                    DisplayOrders(ordersCopy);

                    break;

                case 3:

                    Console.WriteLine("\nSorted using Quick Sort:");

                    QuickSort(ordersCopy, 0, ordersCopy.Length - 1);

                    DisplayOrders(ordersCopy);

                    break;

                case 4:

                    Console.WriteLine("Exiting program.");

                    return;

                default:

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

                    break;

            }

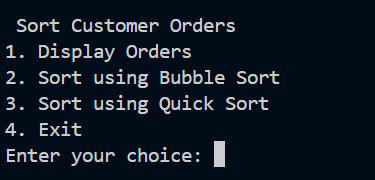
        }

    }

}

**OUTPUT**

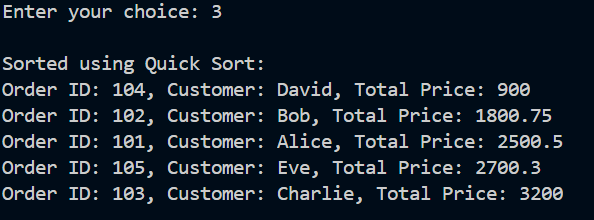
Menu



Sort using Bubble sort



Sort using Quick Sort



**Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| Sorting Algorithm | Best Case | Average Case | Worst Case |
| Bubble Sort | O(n) (When the list is already sorted) | O(n2) | O(n2) (When the list is sorted in reverse order) |
| Quick Sort | O(n log n) (Occurs when the picot divides the Array in equal halves) | O(n log n) | O(n2) (Occurs when the Largest or the smallest element is always chosen as the pivot) |

Quick Sort is generally preffered over bubble sort because:-

1. **Performance-** Bubble Sort has an average time complecityt of O(n2) while quick sort has an average time complexity of O(n log n).
2. **Scalability-** Bubble sort becomes extremely slow when the size of the list increases. Quick Sort handles millions of items efficiently in a fraction of time.
3. **Optimized Variants** Quick sort has several optimized variants like:

* Randomized Quick Sort
* 3-Way Quick Sort

These variants reduces the chances of worst case time complexity and makes Quick Sort even faster