**Exercise 1: Inventory Management System**

In a warehouse system:

* You often need to **add**, **search**, **update**, and **delete** product entries.
* Without efficient structures, operations become **slow and error-prone** as inventory scales.
* Choosing the right data structure (like Dictionary) allows for **constant-time lookups**.

We'll use:

* Dictionary<int, Product> → because ProductId is a unique key.
* This allows for O(1) average time complexity for all operations.

using System;

using System.Collections.Generic;

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 id, string name, int qty, double price)

{

ProductId = id;

ProductName = name;

Quantity = qty;

Price = price;

}

}

public class InventoryManager

{

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

public void AddProduct(Product product)

{

if (!inventory.ContainsKey(product.ProductId))

{

inventory[product.ProductId] = product;

Console.WriteLine($"Product {product.ProductName} added.");

}

else

{

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

}

}

public void UpdateProduct(Product product)

{

if (inventory.ContainsKey(product.ProductId))

{

inventory[product.ProductId] = product;

Console.WriteLine($"Product {product.ProductName} updated.");

}

else

{

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

}

}

public void DeleteProduct(int id)

{

if (inventory.Remove(id))

{

Console.WriteLine($"Product {id} deleted.");

}

else

{

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

}

}

public void DisplayAll()

{

Console.WriteLine("\nInventory:");

foreach (var p in inventory.Values)

{

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

}

}

public Product GetProduct(int id)

{

inventory.TryGetValue(id, out Product product);

return product;

}

}

public class InventoryTest

{

public static void Main(string[] args)

{

InventoryManager manager = new InventoryManager();

var p1 = new Product(101, "Monitor", 15, 10500);

var p2 = new Product(102, "Keyboard", 25, 1500);

var p3 = new Product(103, "Mouse", 40, 800);

manager.AddProduct(p1);

manager.AddProduct(p2);

manager.AddProduct(p3);

manager.DisplayAll();

Console.WriteLine("\nUpdating Product 102:");

var updatedKeyboard = new Product(102, "Mechanical Keyboard", 30, 2500);

manager.UpdateProduct(updatedKeyboard);

Console.WriteLine("\nDeleting Product 101:");

manager.DeleteProduct(101);

manager.DisplayAll();

Console.WriteLine("\nSearching Product 103:");

var result = manager.GetProduct(103);

Console.WriteLine(result != null ? $"Found: {result.ProductName}" : "Product not found");

}

}

Operation Big O Time Complexity Description

AddProduct() O(1) average, O(n) worst Inserts into Dictionary

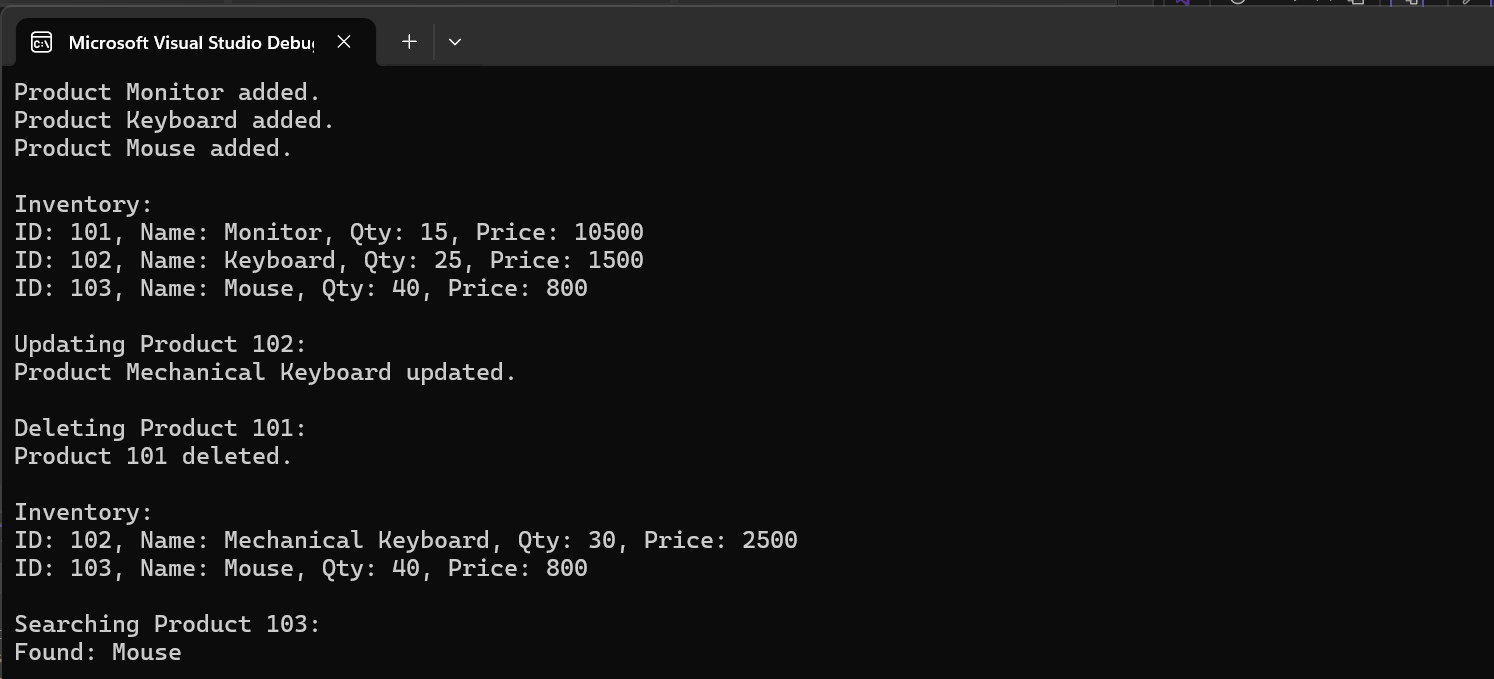
UpdateProduct() O(1) average Replaces value by key

DeleteProduct() O(1) average Removes by key

GetProduct() O(1) average Direct key access

DisplayAll() O(n) Iterates over all entries

**OUTPUT**



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

Big O notation expresses the **upper bound** on the time or space complexity of an algorithm. It allows us to:

* Predict performance as data scales.
* Compare algorithms independently of hardware.
* Choose efficient solutions for large-scale systems.

Scenario | Linear Search (O(n)) | Binary Search (O(log n))

---------------------------------------------------------------------------

Best Case | Target is the first element → O(1) | Target is at the middle → O(1)

Average Case | Element is in the middle → O(n/2) | Midpoint updates log steps → O(log n)

Worst Case | Element not present → O(n) | Max log₂(n) comparisons → O(log n)

public class Product : IComparable<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 int CompareTo(Product other)

{

return this.ProductId.CompareTo(other.ProductId);

}

}

public class ProductSearch

{

// Linear Search: O(n)

public static Product LinearSearch(Product[] products, int targetId)

{

foreach (var product in products)

{

if (product.ProductId == targetId)

return product;

}

return null;

}

// Binary Search: O(log n) — assumes sorted input

public static Product BinarySearch(Product[] sortedProducts, int targetId)

{

int left = 0;

int right = sortedProducts.Length - 1;

while (left <= right)

{

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

if (sortedProducts[mid].ProductId == targetId)

return sortedProducts[mid];

else if (sortedProducts[mid].ProductId < targetId)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

}

public class SearchTest

{

public static void Main(string[] args)

{

Product[] products = {

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

new Product(157, "Headphones", "Electronics"),

new Product(121, "Keyboard", "Electronics"),

new Product(199, "Book", "Books"),

new Product(135, "T-Shirt", "Clothing"),

new Product(105, "Mouse", "Electronics"),

new Product(142, "Coffee Mug", "Kitchenware")

};

Console.WriteLine("Linear Search for Product ID 135");

Product found1 = ProductSearch.LinearSearch(products, 135);

Console.WriteLine($"Result: {(found1 != null ? found1.ProductName : "Not Found")}");

Array.Sort(products); // Required for binary search

Console.WriteLine("\n Binary Search for Product ID 135");

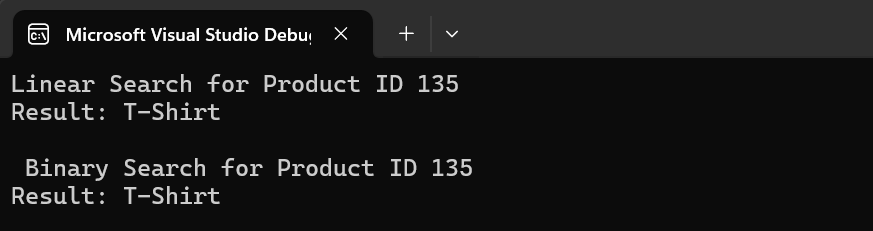
Product found2 = ProductSearch.BinarySearch(products, 135);

Console.WriteLine($"Result: {(found2 != null ? found2.ProductName : "Not Found")}");

}

}

**OUTPUT**



Operation | Linear Search | Binary Search

------------------------------------------------------------

Best Case | O(1) | O(1)

Average Case | O(n) | O(log n)

Worst Case | O(n) | O(log n)

Requirements | Unsorted array | Sorted array

Suitability for E-commerce Platforms

Feature | Linear Search | Binary Search

-----------------------------------------------------------------------------------------------------------

Large-scale Inventory | Slower as size increases (O(n)) | Fast and scalable (O(log n))

Performance | Degrades with dataset growth | Efficient even with big data

Use Case | Good for small lists or debug | Ideal for production systems

**Exercise 7: Financial Forecasting**

Recursion is a technique in which a function **calls itself** to solve a smaller instance of the same problem.

Components of Recursion:

1. Base Case : Stops the recursion to prevent infinite calls.

2. Recursive Step: Calls the same function with a smaller/simpler input.

using System;

public class FinancialForecasting

{

// Recursive method: O(n) time, O(n) space (stack calls)

public static double CalculateFutureValueRecursive(double currentValue, double growthRate, int periods)

{

if (periods == 0)

return currentValue;

double nextValue = currentValue \* (1 + growthRate);

return CalculateFutureValueRecursive(nextValue, growthRate, periods - 1);

}

// Optimized iterative method: O(n) time, O(1) space

public static double CalculateFutureValueIterative(double initialValue, double growthRate, int periods)

{

double futureValue = initialValue;

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

{

futureValue \*= (1 + growthRate);

}

return futureValue;

}

}

public class ForecastTest

{

public static void Main(string[] args)

{

double initialInvestment = 1000.00;

double annualGrowthRate = 0.05; // 5%

int yearsToForecast = 10;

Console.WriteLine("Financial Forecast");

Console.WriteLine($"Initial Investment: ${initialInvestment:N2}");

Console.WriteLine($"Annual Growth Rate: {annualGrowthRate:P1}");

Console.WriteLine($"Forecast Period: {yearsToForecast} years\n");

Console.WriteLine("Using Recursive Method");

double futureRecursive = FinancialForecasting.CalculateFutureValueRecursive(initialInvestment, annualGrowthRate, yearsToForecast);

Console.WriteLine($"Predicted Future Value: ${futureRecursive:N2}");

Console.WriteLine("\nUsing Optimized Iterative Method");

double futureIterative = FinancialForecasting.CalculateFutureValueIterative(initialInvestment, annualGrowthRate, yearsToForecast);

Console.WriteLine($"Predicted Future Value: ${futureIterative:N2}");

}

}

**Time Complexity of the Recursive Algorithm**

Algorithm | Time Complexity | Space Complexity | Notes

------------------------------------------------------------------------------------------------------------------------------------

CalculateFutureValueRecursive() | O(n) | O(n) | Due to stack frames

CalculateFutureValueIterative() | O(n) | O(1) | Constant memory, faster

Optimization Discussion

Recursive Drawbacks:

- Each recursive call uses extra memory on the stack.

- Performance suffers due to call overhead.

- May cause stack overflow for large values of `n`.

Iterative Advantage:

- Avoids recursion depth.

- Requires constant memory.

- Executes faster in real-world scenarios despite same O(n) time.

**OUTPUT**

