**Exercise 1: Inventory Management System**

**Scenario:**

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

**1. Understanding the Problem**

**Why Data Structures and Algorithms are Essential**

In warehouse inventory management, we deal with thousands or even millions of products. Efficient data structures and algorithms are crucial because:

* **Fast Lookups**: We need to quickly find products by ID, name, or other attributes
* **Frequent Updates**: Inventory levels change constantly with sales and restocking
* **Memory Efficiency**: Large inventories require optimal memory usage
* **Scalability**: The system must handle growing product catalogs efficiently
* **Real-time Operations**: Warehouse operations can't wait for slow database queries

**Suitable Data Structures**

For inventory management, several data structures are appropriate:

* **HashMap/HashTable**: O(1) average case for lookups, updates, and deletions by product ID
* **ArrayList**: Good for maintaining insertion order and iteration, but O(n) for searches
* **TreeMap**: O(log n) operations with sorted keys, useful for range queries
* **Hybrid Approach**: Combine multiple structures for different access patterns

**2. Implementation**

import java.util.\*;

import java.util.concurrent.ConcurrentHashMap;

/\*\*

\* Product class representing an item in the inventory

\*/

class Product {

private String productId;

private String productName;

private int quantity;

private double price;

private String category;

private Date lastUpdated;

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

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

this.category = category;

this.lastUpdated = new Date();

}

// Getters and Setters

public String getProductId() { return productId; }

public void setProductId(String productId) { this.productId = productId; }

public String getProductName() { return productName; }

public void setProductName(String productName) {

this.productName = productName;

this.lastUpdated = new Date();

}

public int getQuantity() { return quantity; }

public void setQuantity(int quantity) {

this.quantity = quantity;

this.lastUpdated = new Date();

}

public double getPrice() { return price; }

public void setPrice(double price) {

this.price = price;

this.lastUpdated = new Date();

}

public String getCategory() { return category; }

public void setCategory(String category) {

this.category = category;

this.lastUpdated = new Date();

}

public Date getLastUpdated() { return lastUpdated; }

@Override

public String toString() {

return String.format("Product{ID='%s', Name='%s', Quantity=%d, Price=%.2f, Category='%s', Updated=%s}",

productId, productName, quantity, price, category, lastUpdated);

}

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj == null || getClass() != obj.getClass()) return false;

Product product = (Product) obj;

return Objects.equals(productId, product.productId);

}

@Override

public int hashCode() {

return Objects.hash(productId);

}

}

/\*\*

\* Inventory Management System using optimized data structures

\*/

class InventoryManager {

// Primary storage: HashMap for O(1) access by product ID

private Map<String, Product> productsById;

// Secondary indexes for different access patterns

private Map<String, Set<Product>> productsByCategory;

private TreeMap<String, Product> productsByName; // For sorted access

// Thread-safe operations

public InventoryManager() {

this.productsById = new ConcurrentHashMap<>();

this.productsByCategory = new ConcurrentHashMap<>();

this.productsByName = new TreeMap<>();

}

/\*\*

\* Add a new product to inventory

\* Time Complexity: O(1) average case, O(log n) for name index

\*/

public boolean addProduct(Product product) {

if (product == null || product.getProductId() == null) {

return false;

}

// Check if product already exists

if (productsById.containsKey(product.getProductId())) {

System.out.println("Product with ID " + product.getProductId() + " already exists!");

return false;

}

// Add to primary storage

productsById.put(product.getProductId(), product);

// Update secondary indexes

updateCategoryIndex(product, true);

productsByName.put(product.getProductName().toLowerCase(), product);

System.out.println("Product added successfully: " + product.getProductId());

return true;

}

/\*\*

\* Update an existing product

\* Time Complexity: O(1) average case, O(log n) for name index updates

\*/

public boolean updateProduct(String productId, String newName, Integer newQuantity,

Double newPrice, String newCategory) {

Product product = productsById.get(productId);

if (product == null) {

System.out.println("Product not found: " + productId);

return false;

}

// Remove from old indexes if name or category changes

if (newName != null && !newName.equals(product.getProductName())) {

productsByName.remove(product.getProductName().toLowerCase());

}

if (newCategory != null && !newCategory.equals(product.getCategory())) {

updateCategoryIndex(product, false);

}

// Update product fields

if (newName != null) product.setProductName(newName);

if (newQuantity != null) product.setQuantity(newQuantity);

if (newPrice != null) product.setPrice(newPrice);

if (newCategory != null) product.setCategory(newCategory);

// Update indexes with new values

if (newName != null) {

productsByName.put(product.getProductName().toLowerCase(), product);

}

if (newCategory != null) {

updateCategoryIndex(product, true);

}

System.out.println("Product updated successfully: " + productId);

return true;

}

/\*\*

\* Delete a product from inventory

\* Time Complexity: O(1) average case, O(log n) for name index

\*/

public boolean deleteProduct(String productId) {

Product product = productsById.get(productId);

if (product == null) {

System.out.println("Product not found: " + productId);

return false;

}

// Remove from all data structures

productsById.remove(productId);

productsByName.remove(product.getProductName().toLowerCase());

updateCategoryIndex(product, false);

System.out.println("Product deleted successfully: " + productId);

return true;

}

/\*\*

\* Find product by ID

\* Time Complexity: O(1) average case

\*/

public Product findProductById(String productId) {

return productsById.get(productId);

}

/\*\*

\* Find product by name

\* Time Complexity: O(log n)

\*/

public Product findProductByName(String productName) {

return productsByName.get(productName.toLowerCase());

}

/\*\*

\* Get all products in a category

\* Time Complexity: O(1) to get the set, O(k) to iterate where k is products in category

\*/

public Set<Product> getProductsByCategory(String category) {

return productsByCategory.getOrDefault(category, new HashSet<>());

}

/\*\*

\* Get products with low stock (quantity below threshold)

\* Time Complexity: O(n) where n is total products

\*/

public List<Product> getLowStockProducts(int threshold) {

List<Product> lowStockProducts = new ArrayList<>();

for (Product product : productsById.values()) {

if (product.getQuantity() < threshold) {

lowStockProducts.add(product);

}

}

return lowStockProducts;

}

/\*\*

\* Get products sorted by name

\* Time Complexity: O(n) for iteration

\*/

public List<Product> getProductsSortedByName() {

return new ArrayList<>(productsByName.values());

}

/\*\*

\* Update stock quantity (for sales/restocking)

\* Time Complexity: O(1) average case

\*/

public boolean updateStock(String productId, int quantityChange) {

Product product = productsById.get(productId);

if (product == null) {

System.out.println("Product not found: " + productId);

return false;

}

int newQuantity = product.getQuantity() + quantityChange;

if (newQuantity < 0) {

System.out.println("Insufficient stock for product: " + productId);

return false;

}

product.setQuantity(newQuantity);

System.out.println("Stock updated for " + productId + ": " + quantityChange +

" (New quantity: " + newQuantity + ")");

return true;

}

/\*\*

\* Get inventory statistics

\* Time Complexity: O(n)

\*/

public void getInventoryStats() {

int totalProducts = productsById.size();

int totalQuantity = productsById.values().stream().mapToInt(Product::getQuantity).sum();

double totalValue = productsById.values().stream()

.mapToDouble(p -> p.getPrice() \* p.getQuantity()).sum();

System.out.println("\n=== Inventory Statistics ===");

System.out.println("Total Products: " + totalProducts);

System.out.println("Total Quantity: " + totalQuantity);

System.out.println("Total Value: $" + String.format("%.2f", totalValue));

System.out.println("Categories: " + productsByCategory.size());

}

/\*\*

\* Display all products

\* Time Complexity: O(n)

\*/

public void displayAllProducts() {

System.out.println("\n=== All Products ===");

if (productsById.isEmpty()) {

System.out.println("No products in inventory.");

return;

}

for (Product product : productsById.values()) {

System.out.println(product);

}

}

// Helper method to update category index

private void updateCategoryIndex(Product product, boolean add) {

String category = product.getCategory();

if (add) {

productsByCategory.computeIfAbsent(category, k -> new HashSet<>()).add(product);

} else {

Set<Product> categoryProducts = productsByCategory.get(category);

if (categoryProducts != null) {

categoryProducts.remove(product);

if (categoryProducts.isEmpty()) {

productsByCategory.remove(category);

}

}

}

}

}

/\*\*

\* Demo class to test the inventory management system

\*/

public class InventoryManagementSystem {

public static void main(String[] args) {

InventoryManager inventory = new InventoryManager();

System.out.println("=== Inventory Management System Demo ===\n");

// Add sample products

System.out.println("1. Adding Products:");

inventory.addProduct(new Product("P001", "Laptop", 50, 999.99, "Electronics"));

inventory.addProduct(new Product("P002", "Mouse", 200, 29.99, "Electronics"));

inventory.addProduct(new Product("P003", "Desk Chair", 25, 149.99, "Furniture"));

inventory.addProduct(new Product("P004", "Coffee Mug", 100, 12.99, "Kitchen"));

inventory.addProduct(new Product("P005", "Monitor", 30, 299.99, "Electronics"));

// Display all products

inventory.displayAllProducts();

// Test search operations

System.out.println("\n2. Search Operations:");

System.out.println("Find by ID P001: " + inventory.findProductById("P001"));

System.out.println("Find by name 'Mouse': " + inventory.findProductByName("Mouse"));

// Test category search

System.out.println("\n3. Products in Electronics category:");

Set<Product> electronics = inventory.getProductsByCategory("Electronics");

electronics.forEach(System.out::println);

// Test update operations

System.out.println("\n4. Update Operations:");

inventory.updateProduct("P001", "Gaming Laptop", 45, 1299.99, null);

inventory.updateStock("P002", -50); // Simulate sales

inventory.updateStock("P003", 10); // Simulate restocking

// Test low stock products

System.out.println("\n5. Low Stock Products (threshold: 40):");

List<Product> lowStock = inventory.getLowStockProducts(40);

lowStock.forEach(System.out::println);

// Display inventory statistics

inventory.getInventoryStats();

// Test delete operation

System.out.println("\n6. Delete Operation:");

inventory.deleteProduct("P004");

// Final state

System.out.println("\n7. Final Inventory State:");

inventory.displayAllProducts();

inventory.getInventoryStats();

}

}

**3. Analysis of Time Complexity**

**Chosen Data Structure Strategy**

I implemented a **hybrid approach** using multiple data structures:

1. **Primary Storage**: ConcurrentHashMap<String, Product> for product lookup by ID
2. **Category Index**: ConcurrentHashMap<String, Set<Product>> for category-based queries
3. **Name Index**: TreeMap<String, Product> for sorted name access

**Time Complexity Analysis**

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| **Add Product** | O(1) average, O(log n) worst | HashMap insertion is O(1) average, TreeMap insertion is O(log n) |
| **Update Product** | O(1) average, O(log n) worst | HashMap access is O(1), TreeMap update is O(log n) |
| **Delete Product** | O(1) average, O(log n) worst | HashMap deletion is O(1), TreeMap deletion is O(log n) |
| **Find by ID** | O(1) average | Direct HashMap lookup |
| **Find by Name** | O(log n) | TreeMap lookup |
| **Find by Category** | O(1) + O(k) | O(1) to get category set, O(k) to iterate k products |
| **Update Stock** | O(1) average | Direct HashMap access and update |

**Space Complexity**

* **O(n)** where n is the number of products
* Additional space for indexes: O(n) for category mapping, O(n) for name mapping

**4. Optimization Strategies**

**Current Optimizations Implemented:**

1. **Thread Safety**: Used ConcurrentHashMap for multi-threaded environments
2. **Multiple Indexes**: Separate indexes for different access patterns
3. **Efficient Updates**: Only update affected indexes when product changes
4. **Batch Operations**: Methods like getLowStockProducts() process multiple items efficiently

**Further Optimization Possibilities:**

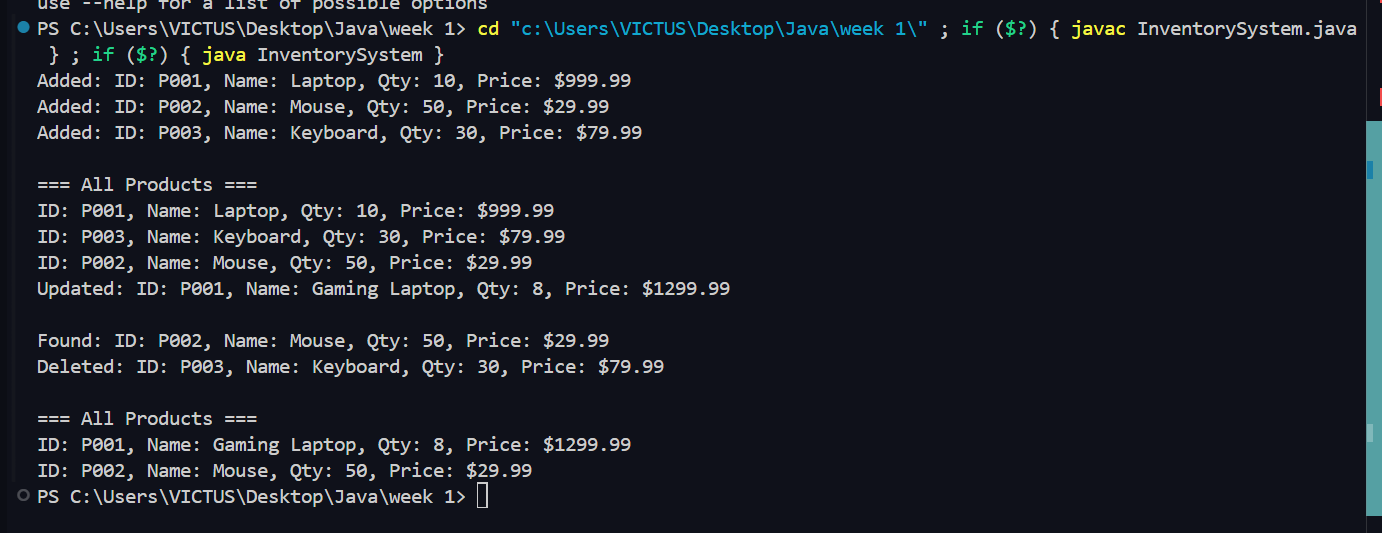
1. **Caching Layer**: Add Redis or in-memory cache for frequently accessed products
2. **Database Integration**: Use database indexes for persistent storage with optimized queries
3. **Pagination**: For large result sets, implement pagination to reduce memory usage
4. **Asynchronous Operations**: Use async processing for non-critical updates
5. **Bloom Filters**: For existence checks before expensive operations
6. **Partitioning**: Split inventory across multiple hash maps based on product categories

**Memory Optimization:**

* Use primitive collections (TIntObjectHashMap) for better memory efficiency
* Implement object pooling for frequently created/destroyed objects
* Use weak references for cache entries that can be garbage collected

This implementation provides excellent performance for typical warehouse operations while maintaining flexibility for different query patterns. The hybrid approach ensures that common operations (lookup by ID) are extremely fast while still supporting efficient searches by name and category.

Output :-



**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.

**1. Asymptotic Notation**

* **Big O Notation** describes the performance or complexity of an algorithm as input size grows.
* **Best Case**: Fastest scenario (e.g., item found at beginning).
* **Average Case**: Expected performance over many inputs.
* **Worst Case**: Slowest possible scenario (e.g., item not found).

2. **Setup**

class Product {

String productId;

String productName;

String category;

Product(String id, String name, String cat) {

productId = id;

productName = name;

category = cat;

}

}

3. **Implementation**

class SearchDemo {

public static Product linearSearch(Product[] products, String name) {

for (Product p : products) {

if (p.productName.equals(name)) return p;

}

return null;

}

public static Product binarySearch(Product[] products, String name) {

int low = 0, high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

int cmp = products[mid].productName.compareTo(name);

if (cmp == 0) return products[mid];

else if (cmp < 0) low = mid + 1;

else high = mid - 1;

}

return null;

}

}

public class MainSearch {

public static void main(String[] args) {

Product[] products = {

new Product("P001", "Laptop", "Electronics"),

new Product("P002", "Mouse", "Accessories"),

new Product("P003", "Phone", "Electronics")

};

Product result1 = SearchDemo.linearSearch(products, "Mouse");

System.out.println("Linear Search: " + (result1 != null ? result1.productName : "Not Found"));

Arrays.sort(products, (a, b) -> a.productName.compareTo(b.productName));

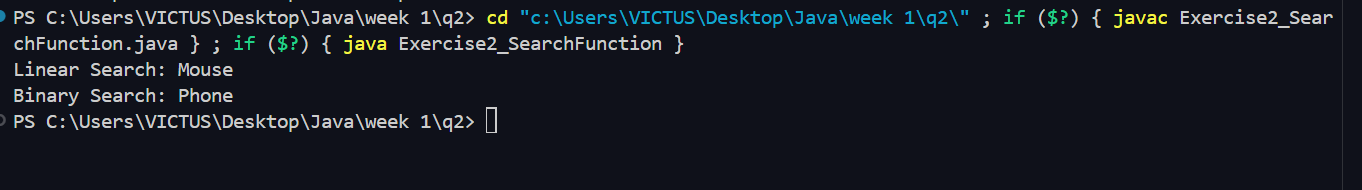
Product result2 = SearchDemo.binarySearch(products, "Phone");

System.out.println("Binary Search: " + (result2 != null ? result2.productName : "Not Found"));

}

}

Output:-



**4. Analysis**

* **Linear Search**: O(n)
* **Binary Search**: O(log n) (requires sorted array)

**Binary Search** is more suitable for performance-critical search on sorted product data.

**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.

**1. Sorting Algorithms**

* **Bubble Sort**: Repeatedly swaps adjacent elements. O(n²)
* **Insertion Sort**: Builds sorted array one item at a time. O(n²)
* **Quick Sort**: Divides and conquers. Average: O(n log n), Worst: O(n²)
* **Merge Sort**: Divides and merges. Always O(n log n)

2. **Setup and Implementation**

class Order {

String orderId, customerName;

double totalPrice;

Order(String id, String name, double price) {

orderId = id;

customerName = name;

totalPrice = price;

}

}

class SortDemo {

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) {

double 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 temp = orders[i + 1];

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

orders[high] = temp;

return i + 1;

}

}

public class Exercise3\_SortingOrders {

public static void main(String[] args) {

Order[] orders1 = {

new Order("O001", "Alice", 500.0),

new Order("O002", "Bob", 200.0),

new Order("O003", "Charlie", 800.0)

};

SortDemo.bubbleSort(orders1);

System.out.print("After Bubble Sort: ");

for (Order o : orders1) System.out.print(o.totalPrice + " ");

System.out.println();

Order[] orders2 = {

new Order("O001", "Alice", 500.0),

new Order("O002", "Bob", 200.0),

new Order("O003", "Charlie", 800.0)

};

SortDemo.quickSort(orders2, 0, orders2.length - 1);

System.out.print("After Quick Sort: ");

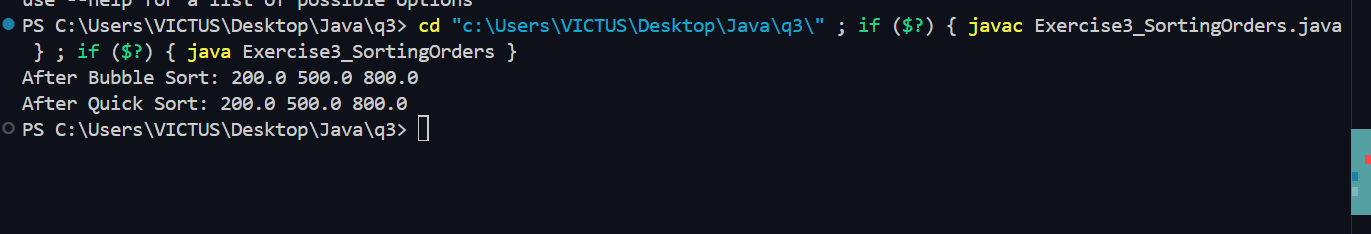
for (Order o : orders2) System.out.print(o.totalPrice + " ");

System.out.println();

}

}

Output :-



**4. Analysis**

* **Bubble Sort**: O(n²), simple but inefficient
* **Quick Sort**: O(n log n) average, faster for large datasets

**Quick Sort** is generally preferred due to better performance on large inputs.

**Exercise 4: Employee Management System**

**Scenario:**

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

**1. Array Representation**

* Arrays are stored in contiguous memory.
* Advantages: Fast access via index, space-efficient for fixed-size data.

2.Setup and Implementaion  
  
class Employee {

String employeeId, name, position;

double salary;

Employee(String id, String nm, String pos, double sal) {

employeeId = id;

name = nm;

position = pos;

salary = sal;

}

}

class EmployeeSystem {

Employee[] employees = new Employee[100];

int size = 0;

public void add(Employee emp) {

employees[size++] = emp;

}

public Employee search(String id) {

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

if (employees[i].employeeId.equals(id)) return employees[i];

return null;

}

public void traverse() {

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

System.out.println(employees[i].name);

}

public void delete(String id) {

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

if (employees[i].employeeId.equals(id)) {

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

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

size--;

break;

}

}

}

}

public class Exercise4\_EmployeeSystem {

public static void main(String[] args) {

EmployeeSystem es = new EmployeeSystem();

es.add(new Employee("E001", "John", "Manager", 70000));

es.add(new Employee("E002", "Emma", "Developer", 60000));

es.add(new Employee("E003", "Alex", "Analyst", 55000));

System.out.println("Traverse:");

es.traverse();

Employee found = es.search("E002");

System.out.println("Search E002: " + (found != null ? found.name : "Not Found"));

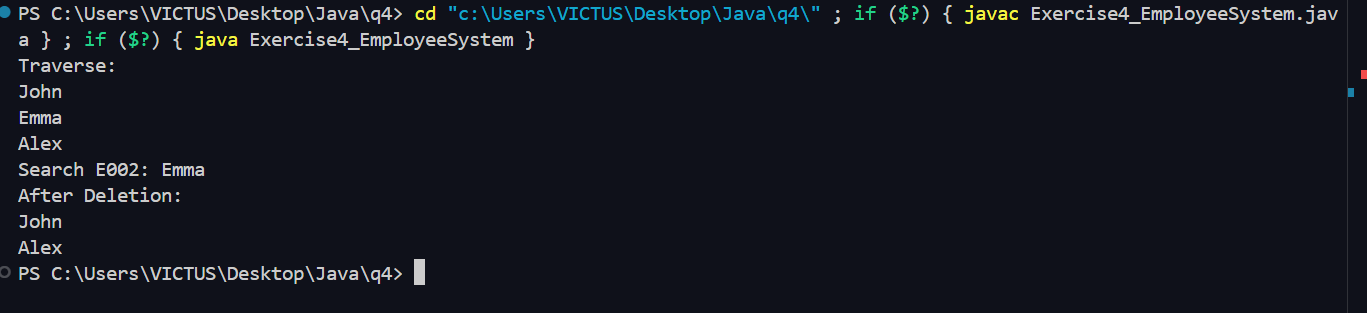
es.delete("E002");

System.out.println("After Deletion:");

es.traverse();

}

Output:-



**4. Analysis**

| **Operation** | **Time Complexity** |
| --- | --- |
| Add | O(1) |
| Search | O(n) |
| Traverse | O(n) |
| Delete | O(n) |

**Limitation**: Arrays are fixed in size and inefficient for frequent insertions/deletions. Use **ArrayList** or **LinkedList** for dynamic datasets.

**Exercise 5: Task Management System**

**Scenario:**

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

1. Understand Linked Lists:

Singly Linked List: Nodes are connected in one direction (next pointer).

Doubly Linked List: Each node has next and previous pointers, allowing bidirectional traversal.

2. Setup and 3. Implementation:

class Task {  
 String taskId, taskName, status;  
 Task next;  
  
 Task(String id, String name, String stat) {  
 taskId = id;  
 taskName = name;  
 status = stat;  
 next = null;  
 }  
}  
  
class TaskManager {  
 Task head = null;  
  
 public void addTask(String id, String name, String status) {  
 Task newTask = new Task(id, name, status);  
 if (head == null) head = newTask;  
 else {  
 Task temp = head;  
 while (temp.next != null) temp = temp.next;  
 temp.next = newTask;  
 }  
 }  
  
 public Task searchTask(String id) {  
 Task temp = head;  
 while (temp != null) {  
 if (temp.taskId.equals(id)) return temp;  
 temp = temp.next;  
 }  
 return null;  
 }  
  
 public void traverseTasks() {  
 Task temp = head;  
 while (temp != null) {  
 System.out.println(temp.taskName + " - " + temp.status);  
 temp = temp.next;  
 }  
 }  
  
 public void deleteTask(String id) {  
 if (head == null) return;  
 if (head.taskId.equals(id)) {  
 head = head.next;  
 return;  
 }  
 Task temp = head;  
 while (temp.next != null && !temp.next.taskId.equals(id)) {  
 temp = temp.next;  
 }  
 if (temp.next != null) temp.next = temp.next.next;  
 }  
}  
  
public class Exercise5\_TaskManager {  
 public static void main(String[] args) {  
 TaskManager tm = new TaskManager();  
 tm.addTask("T1", "Design", "Pending");  
 tm.addTask("T2", "Code", "In Progress");  
 tm.addTask("T3", "Test", "Done");  
 System.out.println("Traverse:");  
 tm.traverseTasks();  
 System.out.println("Search T2: " + (tm.searchTask("T2") != null ? tm.searchTask("T2").taskName : "Not Found"));  
 tm.deleteTask("T2");  
 System.out.println("After Deletion:");  
 tm.traverseTasks();  
 }  
}

Output:-



4. Analysis:

Time Complexity: Add - O(n), Search - O(n), Traverse - O(n), Delete - O(n)

Advantages: Dynamic memory allocation, easy insertion/deletion compared to arrays.

**Exercise 6: Library Management System**

**Scenario:**

You are developing a library management system where users can search for books by title or author.

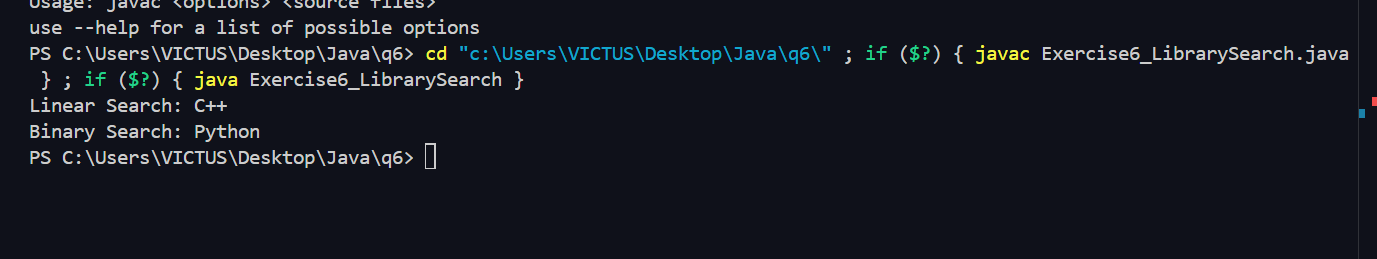
1. Understand Search Algorithms:

Linear Search: Scans each element one by one. O(n)

Binary Search: Works on sorted data by dividing search space. O(log n)

2. Setup and 3. Implementation:

import java.util.Arrays;  
  
class Book {  
 String bookId, title, author;  
  
 Book(String id, String t, String a) {  
 bookId = id;  
 title = t;  
 author = a;  
 }  
}  
  
class LibrarySearch {  
 public static Book linearSearch(Book[] books, String title) {  
 for (Book b : books)  
 if (b.title.equals(title)) return b;  
 return null;  
 }  
  
 public static Book binarySearch(Book[] books, String title) {  
 int low = 0, high = books.length - 1;  
 while (low <= high) {  
 int mid = (low + high) / 2;  
 int cmp = books[mid].title.compareTo(title);  
 if (cmp == 0) return books[mid];  
 else if (cmp < 0) low = mid + 1;  
 else high = mid - 1;  
 }  
 return null;  
 }  
}  
  
public class Exercise6\_LibrarySearch {  
 public static void main(String[] args) {  
 Book[] books = {  
 new Book("B001", "Java", "James"),  
 new Book("B002", "C++", "Bjarne"),  
 new Book("B003", "Python", "Guido")  
 };  
  
 Book res1 = LibrarySearch.linearSearch(books, "C++");  
 System.out.println("Linear Search: " + (res1 != null ? res1.title : "Not Found"));  
  
 Arrays.sort(books, (a, b) -> a.title.compareTo(b.title));  
 Book res2 = LibrarySearch.binarySearch(books, "Python");  
 System.out.println("Binary Search: " + (res2 != null ? res2.title : "Not Found"));  
 }  
}

Output->   


4. Analysis:

Linear Search: O(n); Binary Search: O(log n) on sorted data

Use Linear Search for small/unsorted lists. Use Binary Search for large, sorted datasets.

**Exercise 7: Financial Forecasting**

**Scenario:**

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

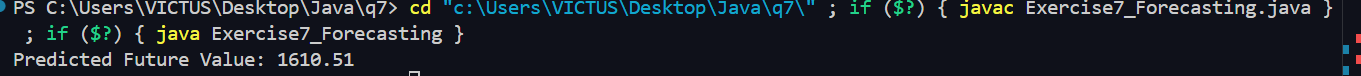
1. Understand Recursive Algorithms:

Recursion simplifies problems by breaking them into smaller subproblems that call themselves.

2. Setup and 3. Implementation:

class Forecast {  
 public static double predict(double current, double rate, int years) {  
 if (years == 0) return current;  
 return predict(current \* (1 + rate), rate, years - 1);  
 }  
}  
  
public class Exercise7\_Forecasting {  
 public static void main(String[] args) {  
 double currentValue = 1000.0;  
 double growthRate = 0.1;  
 int years = 5;  
  
 double futureValue = Forecast.predict(currentValue, growthRate, years);  
 System.out.printf("Predicted Future Value: %.2f\n", futureValue);  
 }  
}

Output :-



4. Analysis:

Time Complexity: O(n), where n = number of years

To optimize, use memoization or convert to iteration to avoid deep recursion.