Inventory Management System with Efficient Data Structures

1. Understanding the Problem

Importance of Data Structures & Algorithms:

* **Efficient Operations**: Large inventories require fast search, insertion, and deletion operations
* **Scalability**: Must handle increasing product counts without performance degradation
* **Real-time Updates**: Inventory changes must be reflected immediately
* **Reporting**: Need quick access for generating reports and analytics

Suitable Data Structures:

1. **HashMap**: O(1) average time complexity for search/insert/delete by product ID
2. **TreeMap**: O(log n) operations with automatic sorting by keys
3. **ArrayList**: Simple but O(n) for search operations
4. **Database**: For persistent storage with indexing capabilities

2. Implementation

import java.util.HashMap;

import java.util.Map;

public class InventoryManagementSystem {

public static void main(String[] args) {

Inventory inventory = new Inventory();

// Add products

inventory.addProduct(new Product("P1001", "Laptop", 50, 999.99));

inventory.addProduct(new Product("P1002", "Smartphone", 200, 699.99));

inventory.addProduct(new Product("P1003", "Headphones", 150, 149.99));

// Update product

inventory.updateProduct("P1001", 45); // Reduce laptop quantity

// Get product

Product p = inventory.getProduct("P1002");

System.out.println("Retrieved product: " + p);

// Generate report

inventory.generateInventoryReport();

// Delete product

inventory.deleteProduct("P1003");

}

static class Product {

private String productId;

private String productName;

private int quantity;

private double price;

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

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

// Getters and setters

public String getProductId() { return productId; }

public String getProductName() { return productName; }

public int getQuantity() { return quantity; }

public void setQuantity(int quantity) { this.quantity = quantity; }

public double getPrice() { return price; }

public void setPrice(double price) { this.price = price; }

@Override

public String toString() {

return String.format("%s: %s (Qty: %d, Price: $%.2f)",

productId, productName, quantity, price);

}

}

static class Inventory {

private Map<String, Product> products;

public Inventory() {

products = new HashMap<>();

}

public void addProduct(Product product) {

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

System.out.println("Added product: " + product);

}

public Product getProduct(String productId) {

return products.get(productId);

}

public void updateProduct(String productId, int newQuantity) {

Product product = products.get(productId);

if (product != null) {

product.setQuantity(newQuantity);

System.out.println("Updated product: " + product);

}

}

public void deleteProduct(String productId) {

Product removed = products.remove(productId);

if (removed != null) {

System.out.println("Removed product: " + removed);

}

}

public void generateInventoryReport() {

System.out.println("\n=== INVENTORY REPORT ===");

System.out.printf("%-10s %-20s %-10s %-10s%n",

"ID", "Name", "Quantity", "Price");

double totalValue = 0;

for (Product p : products.values()) {

System.out.printf("%-10s %-20s %-10d $%-10.2f%n",

p.getProductId(), p.getProductName(),

p.getQuantity(), p.getPrice());

totalValue += p.getQuantity() \* p.getPrice();

}

System.out.printf("%nTotal inventory value: $%.2f%n", totalValue);

}

}

}

3. Analysis and Optimization

Time Complexity Analysis:

| **Operation** | **HashMap** | **ArrayList** | **TreeMap** |
| --- | --- | --- | --- |
| Add Product | O(1) | O(1)\* | O(log n) |
| Get Product | O(1) | O(n) | O(log n) |
| Update Product | O(1) | O(n) | O(log n) |
| Delete Product | O(1) | O(n) | O(log n) |
| Generate Report | O(n) | O(n) | O(n) |

\*O(1) for adding at end, but may be O(n) if resizing needed

Optimization Strategies:

1. **Choose HashMap** for fastest access by product ID
2. **Add Secondary Indexes** for other search criteria (name, category)

private Map<String, Product> nameIndex; // Index by product name

1. **Implement Caching** for frequently accessed products
2. **Batch Processing** for bulk operations
3. **Persistence Layer** with database for large inventories
4. **ConcurrentHashMap** for thread-safe operations in web applications

Output:

