Step 1: Understanding the Problem

Data structures and algorithms are essential in handling large inventories because they enable efficient storage, retrieval, and manipulation of data. In an inventory management system, we need to store information about products, such as their IDs, names, quantities, and prices. We also need to perform operations like adding, updating, and deleting products from the inventory.

The types of data structures suitable for this problem are:

• ArrayList: A resizable array implementation that provides fast access and modification of elements.

• HashMap: A hash table implementation that provides fast lookup, insertion, and deletion of elements based on their keys.

Step 2 and 3: Setup and Implementation

import java.util.HashMap;

import java.util.Map;

// Define a class Product with attributes like productId, productName, quantity, and price

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 void setProductId(String productId) {

this.productId = productId;

}

public String getProductName() {

return productName;

}

public void setProductName(String productName) {

this.productName = 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;

}

}

// Inventory class using HashMap to store products

class Inventory {

private Map<String, Product> products;

public Inventory() {

this.products = new HashMap<>();

}

// Method to add a product to the inventory

public void addProduct(Product product) {

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

}

// Method to update a product in the inventory

public void updateProduct(String productId, Product updatedProduct) {

if (products.containsKey(productId)) {

products.put(productId, updatedProduct);

}

}

// Method to delete a product from the inventory

public void deleteProduct(String productId) {

if (products.containsKey(productId)) {

products.remove(productId);

}

}

// Method to get a product from the inventory

public Product getProduct(String productId) {

return products.get(productId);

}

}

public class Main {

public static void main(String[] args) {

Inventory inventory = new Inventory();

// Create products

Product product1 = new Product("P001", "Product 1", 10, 9.99);

Product product2 = new Product("P002", "Product 2", 20, 19.99);

// Add products to the inventory

inventory.addProduct(product1);

inventory.addProduct(product2);

// Update a product in the inventory

Product updatedProduct1 = new Product("P001", "Updated Product 1", 15, 10.99);

inventory.updateProduct("P001", updatedProduct1);

// Delete a product from the inventory

inventory.deleteProduct("P002");

// Get a product from the inventory

Product retrievedProduct = inventory.getProduct("P001");

System.out.println("Product ID: " + retrievedProduct.getProductId());

System.out.println("Product Name: " + retrievedProduct.getProductName());

System.out.println("Quantity: " + retrievedProduct.getQuantity());

System.out.println("Price: " + retrievedProduct.getPrice());

}

}

Step 4: Analysis

The time complexity of each operation in the HashMap implementation is:

• Add: O(1) average case, O(n) worst case (when hash collisions occur)

• Update: O(1) average case, O(n) worst case (when hash collisions occur)

• Delete: O(1) average case, O(n) worst case (when hash collisions occur)

• Get: O(1) average case, O(n) worst case (when hash collisions occur)

To optimize these operations, we can:

• Use a good hash function to minimize hash collisions

• Use a load factor to ensure the hash table is not too full or too empty

• Use a resizing strategy to adjust the size of the hash table based on the number of elements

• Consider using a different data structure, such as a balanced binary search tree, if the hash table is not suitable for the specific use case.