**Detailed Test Analysis**

The following table summarizes each test, its purpose, and the interpretation of the output:

| **Test Number** | **Description** | **Output** | **Interpretation** |
| --- | --- | --- | --- |
| Test 1 | Adding product P001 | Add product P001: Success | The system successfully added product P001, indicating valid input handling. |
| Test 2 | Adding product with negative quantity | Error adding product: Quantity, volume, and turnover must be non-negative. Add product P002: Failed | The system correctly rejected P002 due to negative quantity, ensuring data integrity. |
| Test 3 | Incoming order for P001 | Process order P001: Success | The system processed an incoming order for P001, likely increasing stock, successfully. |
| Test 4 | Outgoing order exceeding stock | Error processing order: Insufficient stock for outgoing order. Process order P001: Failed | The system prevented an outgoing order that would exceed stock, maintaining inventory accuracy. |
| Test 5 | Space optimization | Warehouse usage: 0.75% Suggestions: [] | Warehouse is 0.75% full, and no optimization suggestions are needed, indicating low utilization. |
| Test 6 | Generating report | Report: [{'product\_id': 'P001', 'quantity': 100}, ... (multiple entries)] | The system generated a report showing multiple quantities for P001, likely tracking transactions or batches. |
| Test 7 | Routing from WH1 to WH3 | Path: ['WH1', 'WH2', 'WH3'] | The system found a route from WH1 to WH3 via WH2, indicating multi-warehouse routing capability. |
| Test 8 | Routing with invalid warehouse | Error routing order: Invalid warehouse ID. Path: [] | The system handled an invalid warehouse ID by returning an error and an empty path, ensuring robustness. |

# inventory\_system.py

import time

from collections import deque

import heapq

import networkx as nx

from typing import Dict, List, Tuple

# Hash table for inventory: {product\_id: {"name": str, "quantity": int, "volume": float, "turnover": float}}

inventory: Dict[str, Dict[str, any]] = {}

# FIFO queue for order processing

order\_queue = deque()

# Priority queue for space optimization (based on turnover rate)

priority\_queue: List[Tuple[float, str]] = []

# Binary Search Tree for sorted reporting

class BSTNode:

    def \_\_init\_\_(self, product\_id, quantity):

        self.product\_id = product\_id

        self.quantity = quantity

        self.left = None

        self.right = None

class BST:

    def \_\_init\_\_(self):

        self.root = None

    def insert(self, product\_id: str, quantity: int):

        self.root = self.\_insert(self.root, product\_id, quantity)

    def \_insert(self, node, product\_id, quantity):

        if not node:

            return BSTNode(product\_id, quantity)

        if product\_id < node.product\_id:

            node.left = self.\_insert(node.left, product\_id, quantity)

        else:

            node.right = self.\_insert(node.right, product\_id, quantity)

        return node

    def sorted\_report(self):

        result = []

        self.\_inorder(self.root, result)

        return result

    def \_inorder(self, node, result):

        if node:

            self.\_inorder(node.left, result)

            result.append({"product\_id": node.product\_id, "quantity": node.quantity})

            self.\_inorder(node.right, result)

bst = BST()

# Graph for warehouse network

warehouse\_graph = nx.Graph()

# Warehouse capacity in cubic meters

WAREHOUSE\_CAPACITY = 10000.0

current\_volume = 0.0

def add\_product(product\_id: str, name: str, quantity: int, volume: float, turnover: float) -> bool:

    """Add or update a product in the inventory."""

    try:

        global current\_volume

        if quantity < 0 or volume < 0 or turnover < 0:

            raise ValueError("Quantity, volume, and turnover must be non-negative.")

        total\_volume = quantity \* volume

        if current\_volume + total\_volume > WAREHOUSE\_CAPACITY:

            raise ValueError("Insufficient warehouse space.")

        inventory[product\_id] = {"name": name, "quantity": quantity, "volume": volume, "turnover": turnover}

        current\_volume += total\_volume

        bst.insert(product\_id, quantity)

        heapq.heappush(priority\_queue, (-turnover, product\_id))  # Negative for max-heap

        return True

    except ValueError as e:

        print(f"Error adding product: {e}")

        return False

def process\_order(product\_id: str, quantity: int, order\_type: str) -> bool:

    """Process incoming or outgoing orders using FIFO queue."""

    try:

        if product\_id not in inventory:

            raise ValueError("Product not found.")

        if quantity < 0:

            raise ValueError("Quantity must be non-negative.")

        if order\_type == "outgoing" and inventory[product\_id]["quantity"] < quantity:

            raise ValueError("Insufficient stock for outgoing order.")

        global current\_volume

        if order\_type == "incoming":

            inventory[product\_id]["quantity"] += quantity

            current\_volume += quantity \* inventory[product\_id]["volume"]

        elif order\_type == "outgoing":

            inventory[product\_id]["quantity"] -= quantity

            current\_volume -= quantity \* inventory[product\_id]["volume"]

        else:

            raise ValueError("Invalid order type. Use 'incoming' or 'outgoing'.")

        order\_queue.append({"order\_id": f"ORD{len(order\_queue) + 1}", "product\_id": product\_id,

                           "quantity": quantity, "type": order\_type, "timestamp": time.time()})

        bst.insert(product\_id, inventory[product\_id]["quantity"])

        return True

    except ValueError as e:

        print(f"Error processing order: {e}")

        return False

def optimize\_space() -> Tuple[float, List[str]]:

    """Calculate space usage and suggest high-turnover products for priority storage."""

    try:

        used\_volume = sum(item["quantity"] \* item["volume"] for item in inventory.values())

        usage\_percentage = (used\_volume / WAREHOUSE\_CAPACITY) \* 100

        suggestions = []

        if usage\_percentage > 80:

            # Suggest prioritizing high-turnover products

            top\_products = heapq.nsmallest(3, priority\_queue)  # Get top 3 by turnover

            suggestions = [f"Prioritize {inventory[pid]['name']} (ID: {pid})" for \_, pid in top\_products]

        return usage\_percentage, suggestions

    except Exception as e:

        print(f"Error in space optimization: {e}")

        return 0.0, []

def generate\_report() -> List[Dict[str, any]]:

    """Generate sorted inventory report using BST."""

    try:

        return bst.sorted\_report()

    except Exception as e:

        print(f"Error generating report: {e}")

        return []

def route\_order(start\_warehouse: str, end\_warehouse: str) -> List[str]:

    """Find optimal shipping route between warehouses using graph."""

    try:

        if not warehouse\_graph.has\_node(start\_warehouse) or not warehouse\_graph.has\_node(end\_warehouse):

            raise ValueError("Invalid warehouse ID.")

        return nx.shortest\_path(warehouse\_graph, start\_warehouse, end\_warehouse)

    except nx.NetworkXNoPath:

        print(f"No path between {start\_warehouse} and {end\_warehouse}")

        return []

    except ValueError as e:

        print(f"Error routing order: {e}")

        return []

# test\_inventory\_system.py

import networkx as nx

def run\_tests():

    print("Running tests...")

    # Initialize warehouse graph

    warehouse\_graph.add\_nodes\_from(["WH1", "WH2", "WH3"])

    warehouse\_graph.add\_edges\_from([("WH1", "WH2"), ("WH2", "WH3")])

    # Test 1: Add product (normal case)

    print("\nTest 1: Adding product P001")

    result = add\_product("P001", "Laptop", 100, 0.5, 10.0)

    print(f"Add product P001: {'Success' if result else 'Failed'}")

    # Test 2: Add product with negative quantity (error case)

    print("\nTest 2: Adding product with negative quantity")

    result = add\_product("P002", "Phone", -10, 0.2, 5.0)

    print(f"Add product P002: {'Success' if result else 'Failed'}")

    # Test 3: Process order (incoming)

    print("\nTest 3: Incoming order for P001")

    result = process\_order("P001", 50, "incoming")

    print(f"Process order P001: {'Success' if result else 'Failed'}")

    # Test 4: Process order (outgoing, insufficient stock)

    print("\nTest 4: Outgoing order exceeding stock")

    result = process\_order("P001", 200, "outgoing")

    print(f"Process order P001: {'Success' if result else 'Failed'}")

    # Test 5: Space optimization

    print("\nTest 5: Space optimization")

    usage, suggestions = optimize\_space()

    print(f"Warehouse usage: {usage:.2f}%")

    print(f"Suggestions: {suggestions}")

    # Test 6: Generate report

    print("\nTest 6: Generating report")

    report = generate\_report()

    print(f"Report: {report}")

    # Test 7: Route order (valid path)

    print("\nTest 7: Routing from WH1 to WH3")

    path = route\_order("WH1", "WH3")

    print(f"Path: {path}")

    # Test 8: Route order (invalid warehouse)

    print("\nTest 8: Routing with invalid warehouse")

    path = route\_order("WH1", "WH4")

    print(f"Path: {path}")

if \_\_name\_\_ == "\_\_main\_\_":

    run\_tests()