**Data Structures and Algorithms using C**

**Case-Based Assignment**

**Assignment Title - Ecommerce Order Fulfillment System**

**Team Members**- Tamanna Gupta & Jatin Gahlot

**Assignment Description**

Develop an e commerce order fulfillment system that manages product inventory, customer orders, and shipping. Use a binary search tree to store product information for efficient searching and a stack to manage order processing and shipping logistics.

**Program Must include-**

Use a binary search tree to efficiently search for products based on product IDs, names, or categories. Utilize a stack to manage order processing and shipping logistics, ensuring timely fulfillment and delivery of customer orders.

**Selecting appropriate Data Structures**

Trees and stacks are fundamental data structures that are well-suited for modeling and solving problems in the domain of order fulfillment system due to their efficiency, simplicity, and suitability for the specific tasks involved:.

1. **Binary Search Trees for Product Inventory:**

* **Efficient Searching:** Binary search trees provide efficient searching capabilities, especially for large datasets like product inventories. Searching for a product by its ID, name, or category can be done in O(log n) time complexity on average, making it suitable for quick retrieval of product information.
* **Ordered Storage:** Binary search trees inherently maintain a sorted order based on the keys (product IDs). This can be useful for tasks like generating reports or displaying products in a sorted manner, enhancing the user experience.
* **Dynamic Operations:** Binary search trees support dynamic operations like insertion, deletion, and searching, which are essential for maintaining and updating the product inventory as new products are added, quantities change, or products are discontinued.

1. **Stacks for Order Processing:**

* **LIFO Behavior:** Stacks follow the Last-In-First-Out (LIFO) principle, which is well-suited for order processing scenarios. Orders are typically processed in the order they are received, and a stack naturally supports this behavior.
* **Simple Implementation:** Stacks are relatively simple to implement and manage. They require only two basic operations: push (to add an order) and pop (to retrieve an order). This simplicity makes them efficient for managing order processing and shipping logistics.
* **Space Efficiency:** Stacks typically use less memory compared to other data structures like queues or lists since they only need to store pointers to the top element. This can be advantageous, especially in memory-constrained environments.

1. **Overall Efficiency and Scalability:**

* By leveraging binary search trees for product inventory and stacks for order processing, the e-commerce system can achieve overall efficiency and scalability.
* Binary search trees offer efficient search operations, while stacks provide streamlined order processing, leading to faster response times and better performance, even as the size of the inventory and the number of orders grow over time.

**Detailed functionalities**

1. **`create\_tree\_node(Product product)`:**

* Description: This function creates a new tree node and initializes it with the given product.
* Working: It allocates memory for a new tree node, assigns the product to the node, and sets its left and right pointers to NULL.

1. **`add\_product\_to\_tree(TreeNode\*\* root, Product product)`:**

* Description: Adds a product to the binary search tree.
* Working: It traverses the tree recursively to find the appropriate position for the new product based on its ID. If the tree is empty, it creates a new node with the product and sets it as the root. If not empty, it compares the product ID with the current node and inserts it into the left subtree if it's less than the current node's ID, otherwise into the right subtree.

1. **`find\_product\_by\_id(TreeNode\* root, int product\_id)`:**

* Description: Searches for a product in the binary search tree by its ID.
* Working: It recursively traverses the tree, comparing the product IDs until a match is found or a leaf node is reached. If a match is found, it returns a pointer to the product; otherwise, it returns NULL.

1. **`inorder\_traversal(TreeNode\* root)`:**

* Description: Performs an in-order traversal of the binary search tree, printing product details.
* Working: It recursively traverses the tree in an in-order manner (left subtree, current node, right subtree), printing the product details of each node as it visits.

1. **`push\_order(Order order, OrderStack\* stack)`:**

* Description: Pushes an order onto the order processing stack.
* Working: It allocates memory for a new stack node, assigns the order to the node, and inserts it at the top of the stack.

1. **`pop\_order(OrderStack\* stack)`:**

* Description: Pops an order from the order processing stack.
* Working: It removes the top node from the stack, retrieves the order, frees the memory allocated for the node, and returns the order.

1. **`is\_empty(OrderStack\* stack)`:**

* Description: Checks if the order processing stack is empty.
* Working: It checks if the top pointer of the stack is NULL, indicating an empty stack.

1. **`process\_order(Order order, ProductInventory\* inventory, OrderStack\* stack)`:**

* Description: Processes an order by updating product quantities and adding it to the order stack.
* Working: It iterates through the products in the order, verifies their availability in the inventory, updates their quantities, calculates the total price of the order, and adds the order to the stack for processing.

1. **`ship\_orders(OrderStack\* stack)`:**

* Description: Ships orders from the order processing stack, printing shipping details.
* Working: It pops orders from the stack one by one and prints shipping details for each order.

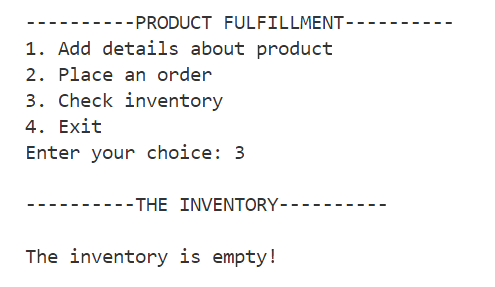
1. **`input\_product\_details(Product \*product)`:**

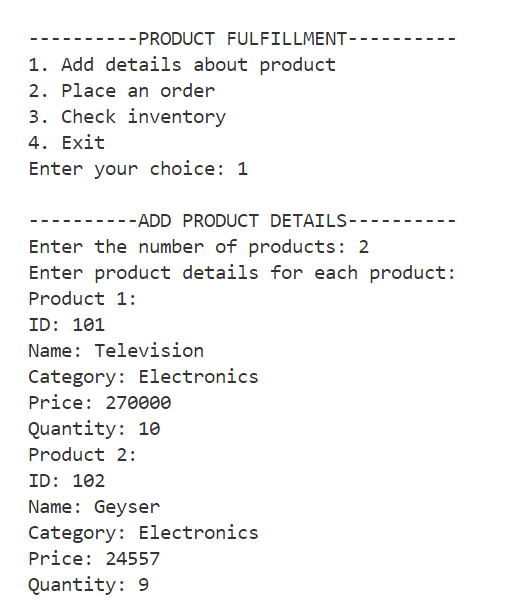
* Description: Inputs details for a product, handling validation for ID, price, and quantity.
* Working: It prompts the user to input details for a product, validates the input for ID, price, and quantity to ensure they are integers or floats, and stores them in the product structure.

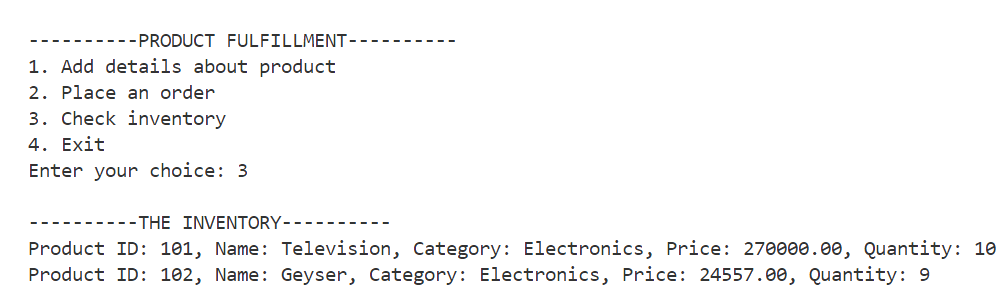
1. **`input\_order\_details(Order \*order)`:**

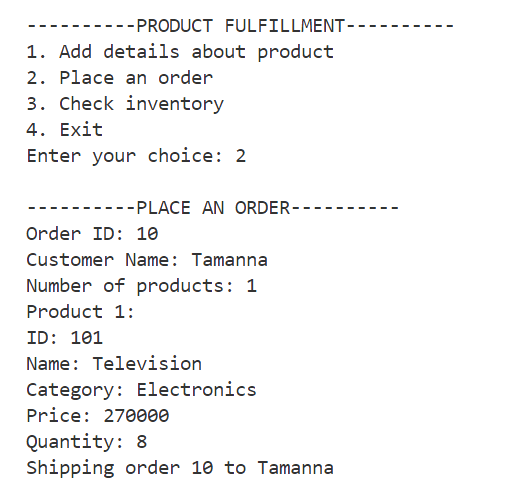
* Description: Inputs details for an order, including product details for multiple products.
* Working: It prompts the user to input details for an order, including order ID, customer name, number of products, and details for each product. It validates the input and dynamically allocates memory for storing product details.

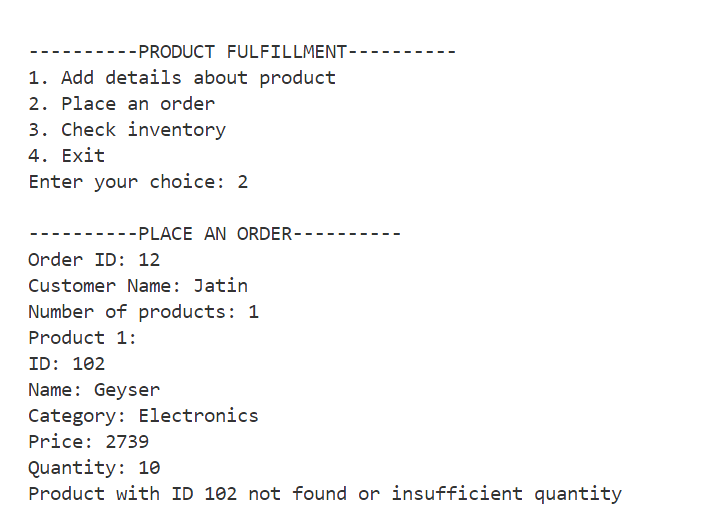
**Output Screen**

****

****

****

****

****