



Project report on Modern Medicine Shop Management System

Course : CSE225

Name	ID
Mahdi Morshed	2221157042

Modern Medicine Shop Management System

Introduction

This lab report presents a comprehensive software solution for managing a modern medicine shop, developed using C++. The program integrates various functionalities required for efficient shop management, including inventory control, customer order processing, and priority-based order handling. The system uses data structures like Stacks, Queues, Priority Queue and Binary Search Trees (BST) to manage different aspects of the shop's operations. The program is divided into four main parts, each handled by different classes and contributors.

Objective

The main objective of this program is to create an efficient and user-friendly system to manage a medicine shop, ensuring that inventory management, customer orders, and priority-based processing are handled seamlessly.

Methodology

1. Manager Entry (Inventory Management)

- **Class:** ManagerEntry
- **Data Structures Used:** Stack

2. Customer Order Processing

- **Class:** CustomerOrder
- **Data Structures Used:** Queue

3. Stored Items Management

- **Class:** Storeditem
- **Data Structures Used:** Binary Search Tree (BST)

4. Priority Queue Order Handling

- **Class:** checkOrder
- **Data Structures Used:** Priority Queue

STACK

Stack has been implemented for the “Storing Products” part of our project. In this part the manager can add and remove items, view the history of actions, display the final state of the shop, and save the inventory to a file.

```
-----  
Storing Product  
-----  
1: Add to Shop  
2: Remove from Shop  
3: Show History  
4: View Final Shop  
0: Exit and Save  
Choose action (0 to exit and save): |
```

There are five options available for the manager

1. Add to Shop: Manager needs to choose option 1 to add products in the shop. Then he needs to give the name and quantity for the newly arrived medicine . Then this item will be saved in the store.

```
-----  
Storing Product  
-----  
1: Add to Shop  
2: Remove from Shop  
3: Show History  
4: View Final Shop  
0: Exit and Save  
Choose action (0 to exit and save): 1  
Enter item name: fexo  
Enter quantity: 100  
  
Choose action (0 to exit and save): 1  
Enter item name: fenadin  
Enter quantity: 120  
  
Choose action (0 to exit and save): 1  
Enter item name: napa  
Enter quantity: 80
```

2. Remove from Shop : If the manager wants to remove any product from the shop or he wants to decrease the quantity then he needs to choose option 2.

```
Choose action (0 to exit and save): 2  
Enter item name: fexo  
Enter quantity: 20  
  
Choose action (0 to exit and save): 2  
Enter item name: fenadin  
Enter quantity: 10
```

3. Show History : If the manager wants to see the history of the shop then he needs to choose option 3. It will be shown according to LIFO.

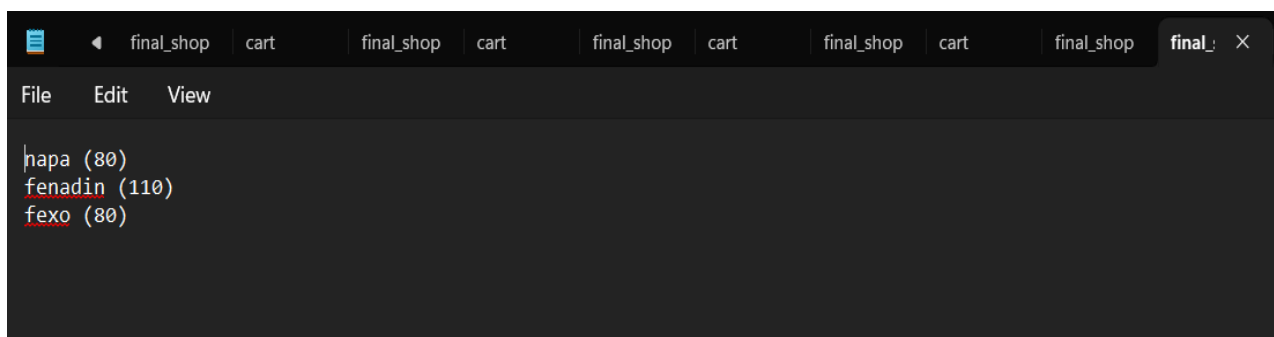
```
-----  
User action history (most recent first):  
-----  
  
Removed: fenadin (10)  
  
Removed: fexo (20)  
  
Added: napa (80)  
  
Added: fenadin (120)  
  
Added: fexo (100)
```

4. View Final Shop : If the manager wants to see the final shop product then he needs to choose option 4. This interface will show all product in the shop according to alphabetical order.

```
-----  
Final shop items:  
-----  
  
Item 1  
Name: napa  
Quantities: (80)  
  
Item 2  
Name: fenadin  
Quantities: (110)  
  
Item 3  
Name: fexo  
Quantities: (80)
```

0. Exit and Save :

If option 0 is chosen then final shop items information will be saved in final_shop.txt file and the programme will return to the main menu.



The screenshot shows a text editor window with multiple tabs. The active tab is labeled 'final_'. The editor displays the following text:

```
napa (80)  
fenadin (110)  
fexo (80)
```

The text is displayed in a monospaced font, with the first line 'napa (80)' on the first line, 'fenadin (110)' on the second line, and 'fexo (80)' on the third line. The editor has a dark background and a light-colored text.

The “Manager Entry” class is designed to manage storing products by allowing the manager to add and remove items, view the history of actions, display the final state of the shop, and save the inventory to a file. We utilize three stacks to manage items, quantities, and the history of actions performed, also three temporary stacks used within the functions. The class also provides a user menu for interaction.

The Stack We Utilizes

- **StackType<string> historyStack:** Stores a history of actions performed.
- **StackType<string> itemStack:** Stores the names of items in the shop.
- **StackType<int> quantityStack:** Stores the quantities of the items.
- **StackType<string> tempItemStack:** Used in both performAddAction and performRemoveAction to temporarily hold items during operations.
- **StackType<int> tempQuantityStack:** Used in both performAddAction and performRemoveAction to temporarily hold quantities during operations.
- **StackType<string> tempStack:** Used in showHistory to temporarily hold history items during display.

The Functions We Use

void performAddAction(const string& itemName, int quantity)

This function adds a specified quantity of an item to the shop's inventory. If the item already exists, the quantity is updated; otherwise, the item is added to the stacks. The action is recorded in the history stack.

```
void performAddAction(const string& itemName, int quantity) {
    StackType<string> tempItemStack;
    StackType<int> tempQuantityStack;
    bool itemFound = false;

    while (!itemStack.IsEmpty()) {
        string currentItem = itemStack.Top();
        int currentQuantity = quantityStack.Top();
        itemStack.Pop();
        quantityStack.Pop();

        if (currentItem == itemName) {
            currentQuantity += quantity;
            itemFound = true;
        }

        tempItemStack.Push(currentItem);
        tempQuantityStack.Push(currentQuantity);
    }

    while (!tempItemStack.IsEmpty()) {
        itemStack.Push(tempItemStack.Top());
        quantityStack.Push(tempQuantityStack.Top());
        tempItemStack.Pop();
        tempQuantityStack.Pop();
    }

    if (!itemFound) {
        if (itemStack.IsFull()) {
            cout << "Shop is full, cannot add more items." << endl;
            return;
        }
        itemStack.Push(itemName);
        quantityStack.Push(quantity);
    }

    string action = "Added: " + itemName + " (" + to_string(quantity) + ")";
    try {
        historyStack.Push(action);
    } catch (FullStacks) {
        cout << "Error: History stack is full." << endl;
    }
}
```

void performRemoveAction(const string& itemName, int quantity)

This function removes a specified quantity of an item from the shop's inventory. If the item exists and there is sufficient quantity, the quantity is updated; if not, an error message is displayed. The action is recorded in the history stack.

```
void performRemoveAction(const string& itemName, int quantity) {
    StackType<string> tempItemStack;
    StackType<int> tempQuantityStack;
    bool itemFound = false;

    while (!itemStack.IsEmpty()) {
        string currentItem = itemStack.Top();
        int currentQuantity = quantityStack.Top();
        itemStack.Pop();
        quantityStack.Pop();

        if (currentItem == itemName) {
            if (currentQuantity >= quantity) {
                currentQuantity -= quantity;
                itemFound = true;
            } else {
                cout << "Item not found in shop or insufficient quantity: " << itemName << endl;
                return;
            }
        }

        if (currentQuantity > 0) {
            tempItemStack.Push(currentItem);
            tempQuantityStack.Push(currentQuantity);
        }
    }

    if (itemFound) {
        itemStack.Push(currentItem);
        quantityStack.Push(currentQuantity);
        removed: " + itemName + " (" + to_string(quantity) + ")";
        Push(action);
    } else {
        cout << "History stack is full." << endl;
    }
}
```

void showHistory()

This function displays the history of all actions performed by the user, with the most recent actions first.

```
void showHistory() {
    StackType<string> tempStack = historyStack;
    cout << "-----" << endl;
    cout << "User action history (most recent first):" << endl;
    cout << "-----" << endl << endl;
    while (!tempStack.IsEmpty()) {
        try {
            string lastAction = tempStack.Top();
            cout << lastAction << endl << endl;
            tempStack.Pop();
        } catch (EmptyStack&) {
            cout << "Error: History stack is empty." << endl;
        }
    }
}
```

void viewFinalShop()

This function displays the final state of the shop's inventory, showing all items and their quantities.

```
void viewFinalShop() {
    StackType<string> tempItemStack = itemStack;
    StackType<int> tempQuantityStack = quantityStack;
    if (tempItemStack.IsEmpty()) {
        cout << "Shop is empty." << endl;
        return;
    }

    cout << "-----" << endl;
    cout << "Final shop items:" << endl;
    cout << "-----" << endl << endl;

    int itemNumber = 1;
    while (!tempItemStack.IsEmpty()) {
        string item = tempItemStack.Top();
        int quantity = tempQuantityStack.Top();
        tempItemStack.Pop();
        tempQuantityStack.Pop();

        cout << "Item " << itemNumber++ << endl;
        cout << "Name: " << item << endl;
        cout << "Quantities: (" << quantity << ")" << endl << endl;
    }
}
```

void save()

This function saves the final state of the shop's inventory to a file named `FINAL_SHOP_FILE`. If the file cannot be opened, an error message is displayed.

```
void save() {
    ofstream finalShopFile(FINAL_SHOP_FILE);
    if (finalShopFile.is_open()) {
        StackType<string> tempItemStack = itemStack;
        StackType<int> tempQuantityStack = quantityStack;
        if (tempItemStack.IsEmpty()) {
            finalShopFile << "Shop is empty." << endl;
        } else {
            while (!tempItemStack.IsEmpty()) {
                finalShopFile << tempItemStack.Top() << " (" << tempQuantityStack.Top() << ")" << endl;
                tempItemStack.Pop();
                tempQuantityStack.Pop();
            }
            finalShopFile.close();
        }
    } else {
        cout << "Unable to open final shop file." << endl;
    }
}
```

void action()

This function displays the menu of available actions to the user.

void menu()

This function provides an interactive menu to the user, allowing them to add or remove items, view the history, display the final shop state, or exit and save the inventory to a file

Queue

Queue has been implemented for the “Customer Order” part of our project. In this part customers can give their order and get services. At first customers need to enter their name and priority.

```
-----  
Customer Inforfation  
-----  
  
Enter name: Nasim  
Enter priority: 25
```

Then there are three options available for the customers

1. Add item to cart : Customers need to choose option 1. Then they need to give the name and quantity of the required medicine for each medicine. If the medicine and required quantity is available in the shop then it will be added to the cart. Otherwise it will show required medicine or quantity is not available in the shop.

```
-----  
Shopping Cart Menu  
-----  
  
1. Add item to cart  
2. Remove item from cart  
3. Display Cart and Exit  
  
Enter your choice: 1  
Enter the item to add to the cart: fexo  
Enter the quantity: 20  
Item added to the cart.
```

```
-----  
Shopping Cart Menu  
-----  
  
1. Add item to cart  
2. Remove item from cart  
3. Display Cart and Exit  
  
Enter your choice: 1  
Enter the item to add to the cart: fenadin  
Enter the quantity: 30  
Sorry, fenadin is not available.
```

```
-----  
Shopping Cart Menu  
-----  
  
1. Add item to cart  
2. Remove item from cart  
3. Display Cart and Exit  
  
Enter your choice: 1  
Enter the item to add to the cart: ace  
Enter the quantity: 120  
Sorry, the required quantity of ace is not available.  
Sorry, ace is not available.
```


2. Remove item from cart : If option 2 is chosen, customers can remove it if any product is entered by mistake. They can remove whatever quantity they want.

```
-----  
Shopping Cart Menu  
-----  
  
1. Add item to cart  
2. Remove item from cart  
3. Display Cart and Exit  
  
Enter your choice: 2  
Enter the item to remove from the cart: fexo  
Enter the quantity to remove: 10  
Removed 10 of fexo from the cart.
```

```
-----  
Shopping Cart Menu  
-----  
  
1. Add item to cart  
2. Remove item from cart  
3. Display Cart and Exit  
  
Enter your choice: 2  
Enter the item to remove from the cart: fenadin  
Enter the quantity to remove: 20  
Item not found in the cart.
```

3. Display Cart and Exit : Customers can see their cart by putting 3 when the program asks them to enter their choice. It will show the name and priority of the customer and then show products name and quantity in a fifo order(Queue). After that customers need to proceed to the counter and get their ordered medicines.

```
-----  
Shopping Cart Menu  
-----  
  
1. Add item to cart  
2. Remove item from cart  
3. Display Cart and Exit  
  
Enter your choice: 3  
  
Customer: Nasim (Priority: 25)  
-----  
Items in the cart:  
-----  
  
fexo (Quantity: 10)  
  
Thanks for using.  
Please proceed to the counter.
```

To implement all the options that are available for the customers we created a class named “CustomerOrder” and made functions to handle different parts.

1. **displayMenu :** Here the options (Menu) is shown to the customers so that users can use the program easily.

2. **addItemToCart** : In this function we take orders from the customer and at first read a txt file named “final_shop” . If the ordered product and quantity is available we take the order and enqueue that in a queue named “cart”. If not then we ask for another order or change quantity.

```
void addItemToCart(const string& filePath) {
    string itemName;
    int quantity;

    cout << "Enter the item to add to the cart: ";
    getline(cin, itemName);
    cout << "Enter the quantity: ";
    cin >> quantity;

    ifstream inFile(filePath);
    if (!inFile) {
        cerr << "Error: Unable to open file for reading." << endl;
        return;
    }

    bool found = false;
    string line;
    while (getline(inFile, line)) {
        if (line.find(itemName) != string::npos) {
            int pos = line.find('(');
            int endPos = line.find(')');
            if (pos != string::npos && endPos != string::npos) {
                int availableQuantity = stoi(line.substr(pos + 1, endPos - pos - 1));
                if (quantity <= availableQuantity) {
                    found = true;
                    string item = itemName + " (Quantity: " + to_string(quantity) + ")";
                    cart.Enqueue(item);
                    cout << "Item added to the cart." << endl;
                } else {
                    cout << "Sorry, the required quantity of " << itemName << " is not available." << endl;
                }
            }
        }
    }
}
```

3. **removeItemFromCart** : To remove an item from the cart a new temporary queue is created to compare and find with the products that has been enqueued to the “cart” queue and if that product is available in the cart then the requested quantity is removed from the cart. Then the new amount of the product is enqueued to “cart”.

```
while (!cart.IsEmpty()) {
    cart.Dequeue(currentItem);
    if (currentItem.find(itemToRemove) != string::npos && !itemFound) {
        itemFound = true;
        int pos = currentItem.find("(Quantity:");
        if (pos != string::npos) {
            int currentQuantity = stoi(currentItem.substr(pos + 11));
            currentQuantity -= removeQuantity;
            if (currentQuantity <= 0) {
                continue;
            } else {
                currentItem = currentItem.substr(0, pos + 11) + to_string(currentQuantity) + ")";
            }
        }
    }
    tempCart.Enqueue(currentItem);
}

if (!itemFound) {
    cout << "Item not found in the cart." << endl;
} else {
    cout << "Removed " << removeQuantity << " of " << itemToRemove << " from the cart." << endl;
}

while (!tempCart.IsEmpty()) {
    tempCart.Dequeue(currentItem);
    cart.Enqueue(currentItem);
}
```

4. **displayCart** : All the products that have been ordered from the customer will be shown by this function along with their name and priority. A file named “cart.txt” will be created and all the orders will be written there.

```
QueType<string> tempCart = cart;
string item;
ofstream outFile;

if (writeToFile) {
    outFile.open("cart.txt", ios::app);
    if (!outFile) {
        cerr << "Unable to open file for writing." << endl;
        return;
    }
    outFile << "Customer: " << customerName << " (Priority: " << priority << ")" << endl;
    outFile << "Items in the cart:" << endl;
}

cout << endl << endl ;
cout << "Customer: " << customerName << " (Priority: " << priority << ")" << endl;
cout << "-----" << endl;
cout << "Items in the cart:" << endl;
cout << "-----" << endl << endl ;

while (!tempCart.IsEmpty()) {
    tempCart.Dequeue(item);
    cout << item << endl;
    if (writeToFile) {
        outFile << item << endl;
    }
}
```

5. **processOrder** : It calls all the functions above according to users demand and runs the program.
6. **cstm** : Takes customers name, priority and calls another function named “setCustomerInfo” to save them. It also calls the “processOrder” function.

Binary Search Tree (BST)

BST has been used to search and manage items in a store inventory efficiently, and it is implemented in the BST.cpp and storeType.cpp files. In terms of the execution of our project, the implementation of BST has been done in the 2nd option (Check Stored Products) of the Manager Entry Interface, when selected, opens up a menu to retrieve product information based on a specific item name and als update it's information.

```
Menu:
1. Search for an item
2. Show all items
3. Exit
Enter your choice: |
```

Search for an item:

Searching for the item by name :

When the user selects option 1 from the Menu given above then the SearchItem function is called after the user enters the name of their desired item, which searches for the item in the BST and allows the user to update its availability if found.

```
Enter the name of the item to search for: fexo
-----
Name            Availability
-----
fexo            80
```

Availability updation:

After the user has successfully searched for a particular item, the program prompts the user if they want to update the availability of an item or not while searching for it. This is done through the SearchItem function. Then the user can enter a new availability value, and the item's availability is updated accordingly.

```
Do you want to update the availability? (1 for Yes, 0 for No): 1
Enter the new availability: 100
Availability updated successfully.
```

Show all items:

When the user selects this option the program displays the entire inventory in a tabular manner by calling the PrintInventory function.

```
Showing all items in inventory:
-----
Name            Availability
-----
fenadin         110
fexo             100
napa             80
```

Exit: The program is terminated when this option is selected.

Classes that interact with the implementation of BST:

The Storeditem class and its functions, readItemsFromFile and displayMenu, are implemented in the main.cpp file. This file serves as the main program file where the functionality of reading items from a file, managing the store inventory using a BST, and providing a user interface for interacting with the inventory are implemented.

Storeditem:

The Storeditem class is designed to manage items in a store inventory. It contains several functions that interact with the BST data structure to manage items, including reading items from a file, displaying the menu, and searching for items.

readItemsFromFile(const string& filename, BST<storeItem>& inventory):

This function reads items from a file and inserts them into the BST.

displayMenu(BST<storeItem>& inventory): This function displays a menu to the user and allows them to interact with the program. The menu includes options to search for an item, show all items, or exit the program. The user's choice is read and then the corresponding action is performed.

bstMain(BST<storeItem>& inventory): This function is the main entry point for the program. It calls readItemsFromFile to read items from a file and then calls displayMenu to display the menu to the user.

storeItem Class

This class represents items in the store inventory. This class is implemented inside the storeType.cpp file. The key functions in this file are:

- storeItem(): This is the default constructor for the storeItem class.
- storeItem(string name, int availability): This is the parameterized constructor for the storeItem class, which initializes the item name and availability.
- getName() const: This function returns the name of the item.
- getAvailability() const: This function returns the availability of the item.
- setAvailability(int avail): This function sets the availability of the item.

BST class

Customs functions in BST class inside the BST.cpp file are:

- PrintTableHeader(): This function prints the header for the inventory table.
- InOrder(TreeNode<ItemType>* tree): This function performs an in-order traversal of the BST and prints each item in the inventory table.
- PrintInventory(): This function prints the entire inventory table is
- SearchItem(string itemName): This function searches for an item in the BST and allows the user to update its availability if found.
- .WriteToFile(const string& filename): This function writes the inventory to a file.
- WriteInventoryToFile(TreeNode<ItemType>* tree, ofstream& outFile): This function recursively writes the inventory to a file.

Priority Queue

In this part we have used priority queue. In that code we have simulated a system for processing customer orders based on priority. Breakdown of this code:

User Struct: The code defines a struct named user to store information about each customer. It includes fields for priority(integer), name(string), order number(integer) and items(strings).

Checkpriority function: This function is the entry point for processing orders. It creates two queues: Bookqueue which is used to store high-priority orders and zeroPriorityQueue which is used to store regular priority orders.

This part belongs to the manager section. At the beginning of this code it opens the file and reads the line one by one. After that it parses the line to get the customer name and priority. It reads subsequent lines until an empty line to collect the list of items for that customer. The priority number was given manually to the customer. Based on the priority, the user is added either bookQueue or zeroPriorityQueue and gets printed out by their priority. In this part we have used heap sort as an algorithm.

```
Nasim with priority 23 is getting the ordered items.
Items in the cart:
fexo (Quantity: 1)
fenadin (Quantity: 5)

Shabbir with priority 10 is getting the ordered items.
Items in the cart:
fexo (Quantity: 10)
fenadin (Quantity: 10)

N.Ahmed with priority 1 is getting the ordered items.
Items in the cart:
fexo (Quantity: 1)

Tasfiq with priority 0 is getting the ordered items.
Items in the cart:
ace (Quantity: 7)

Nas with priority 0 is getting the ordered items.
Items in the cart:
ace (Quantity: 9)
```

Algorithms Implemented

- Recursion.
- In-Order Traversal
- Predecessor Search
- Heapsort

Results

The program successfully integrates the following functionalities:

- **Inventory Management:** The manager can add, remove, and view items in the inventory, with a history of actions and a final state view.
- **Customer Order Processing:** Customers can interactively add and remove items from their cart, with a final display of their cart contents.
- **Priority-Based Order Handling:** Customer orders are processed based on priority, ensuring that higher priority orders are handled first.
- **Item Storage and Retrieval:** Items are efficiently stored and retrieved using a Binary Search Tree, allowing for quick search and display of all items.

Conclusion

The Modern Medicine Shop Management System demonstrates an effective integration of various data structures to handle complex shop management tasks. The use of Stacks, Queues, Priority Queues, and BSTs ensures that the program is both efficient and scalable. Each component works together to provide a seamless user experience for both shop managers and customers.