



UTM
UNIVERSITI TEKNOLOGI MALAYSIA

Department of Computer Science
Faculty of Computing

DATA STRUCTURE AND ALGORITHMS PROJECT

Programme	: Bachelor of Computer Science (<i>Data Engineering</i>)
Subject Code	: SECJ2013
Session-Sem	: 2023/2024-1

Prepared by :
1) LOO JIA CHANG (A22EC0074)
2) GOH JING YANG (A22EC0052)
3) LOW JIE SHENG (A22EC0075)

Section : 02

Group : Nothing

Topic : Inventory Management System

Lecturer : Dr. LIZAWATI BINTI MI YUSUF

Date : 18/1/2023

TABLE OF CONTENTS

1.0 Objective

2.0 Synopsis

3.0 Problem analysis

4.0 Class Diagram

5.0 Pseudo Code

6.0 Description of how data structure operations

7.0 User manual or guide

1.0 Objective

The objective of this project is to develop a Warehouse Inventory Management System using the different concepts of data structure algorithms in C++. The system is designed to efficiently manage and perform operations on a collection of goods/items within the warehouse. The program employs a queueing algorithm to store and manipulate the data related to goods, providing functionalities such as importing items from a file, adding new items, removing items, displaying the inventory, and a stack algorithm for printing a history of actions.

1. FIFO inventory storing algorithm

The first in first out (FIFO) concept ensures that the oldest items in the inventory are the first to be used or shipped. This helps in rotating stock efficiently, preventing perishable goods from expiring or becoming obsolete. It's particularly important for industries where products have a limited shelf life.

2. Neat and Creative Output:

Develop a user-friendly and visually appealing interface for displaying inventory and history.

Ensure clear and descriptive messages for user interactions.

3. Data Hiding and Encapsulation:

Apply the concepts of data hiding and encapsulation by making the goods class private and providing public methods for accessing attributes.

4. History Tracking:

Maintain a history of actions performed, including additions and removals, and save this history to a file when exiting the system.

5. User Interaction:

Provide a menu-driven interface for users to interact with the system, making it intuitive and easy to use.

6. File Operations:

Enable the system to import goods information from a file, export the result after operation and save the history to a file.

This project aims to demonstrate effective implementation of linked lists, encapsulation, user interface design, and file operations in a practical application of Warehouse Inventory Management.

2.0 Synopsis

The Inventory Management System is designed to productively oversee and perform operations on a collection of products or items inside the stockroom because it is basic for eros businesses or organizations to track or oversee their stock. By utilizing our system, users can oversee the lifecycle of inventory items from the minute they are imported into the system until they take off the distribution center. The system encourages the expansion and evacuation of items through an natural interface, supported by a solid line for item management and a stack for keeping up a brief history of exchanges. Each thing are spoken to as a question with unmistakable properties such as ID, name, price, and location, ensuring clear distinguishing proof and following. History includes permits for the review examination of stock changes, which is basic for inspecting and detailing purposes. Generally, the system points to streamlining stockroom operations, decreasing blunders in stock administration, and giving fast get-to-item and exchange information, making it an important device for any inventory-reliant business.

3.0 Problem analysis Problem Analysis

1.Manual System Inefficiencies:

- The existing manual inventory system does not has automation, which results in inefficiencies when it comes to managing stockroom operations.
- Manual procedures notorious for their time-consuming nature and tendency to introduce errors, ultimately impacting overall efficiency and productivity.

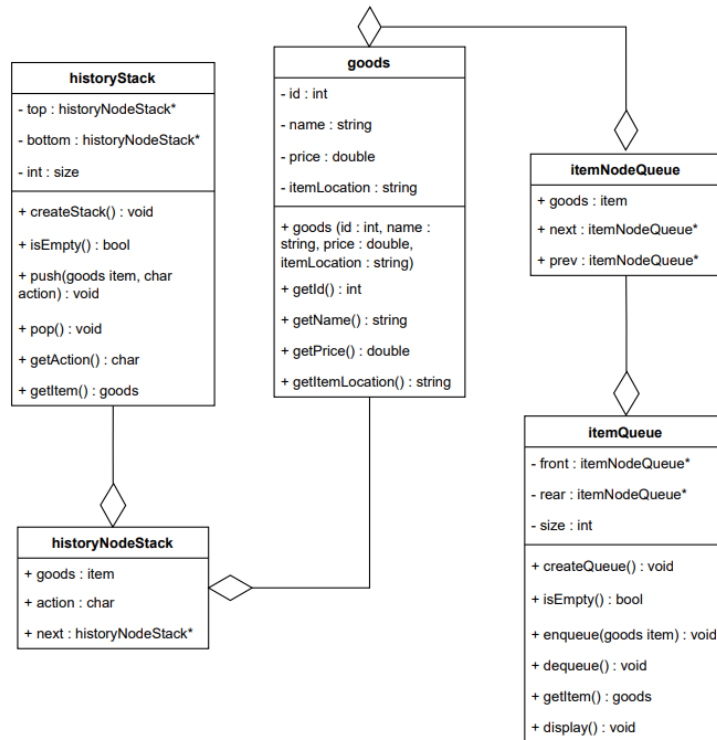
2.Resource Optimization Issues:

- The absence of automated systems can lead to inadequate utilization of resources.
- Managing inventory and making strategic decisions for resource allocation can be challenging in terms of tracking movement and obtaining accurate information.

3.Error-Prone Operations:

- Rephrase The manual input of data contributes to a higher probability of inaccuracies when recording and maintaining inventory data.
- Inaccurate information has the potential to give rise to problems like stock shortages or excessive inventory levels.

4.0 Class Diagram



5.0 Pseudo Code

Main and other function:

Function main ()

1. Display welcome screen.
2. Create an empty itemQueue for managing the inventory.
3. Create an empty historyStack for recording actions.
4. Enter a loop to display the menu and handle user choices:
 - a. Display the header.
 - b. Display the menu options.
 - c. Get the user's choice (integer).
 - d. Validate the user's input using isNumber function.
 - e. If the input is not a valid integer, show an error message and continue to the menu.
 - f. Perform actions based on the user's choice:
 - If choice is 1:
 - Call import function to load items from a file.
 - Display import status.
 - If choice is 2:
 - Call add function to get item details from the user.
 - Enqueue the item to the itemQueue.
 - Push an 'add' action to the historyStack.
 - Display "Item added!" message.
 - If choice is 3:
 - Check if the itemQueue is empty.
 - If it's empty, display "No item in the inventory!" message.
 - Otherwise:
 - Dequeue an item from the itemQueue.
 - Push a 'remove' action to the historyStack.
 - Display "Item removed!" message.
 - If choice is 4:

- Check if the itemQueue is empty.
- If it's empty, display "No item in the inventory!" message.
- Otherwise:
 - Display items from the itemQueue.
- If choice is 5:
 - Call printHistory function to print history to a file.
 - Display "Printing history..." message.
 - Display "History saved into file" message.
 - Display "Exiting..." message.
 - Sleep for 2 seconds.
 - Clear the console screen.
 - End the program.

g. Pause the program for user interaction.

5. End the program.

DisplayHeader Function:

1. Clear the console screen.
2. Display the program header.

Menu Function:

1. Display the available menu options.

IsNumber Function (input validation for integers):

1. Check if the input string consists of digits only.
2. Return true if it's a valid integer, false otherwise.

IsFloat Function (input validation for floating-point numbers):

1. Check if the input string is a valid floating-point number.
2. Return true if it's a valid float, false otherwise.

WelcomeScreen Function:

1. Clear the console screen.
2. Display a welcome message.
3. Wait for user confirmation (e.g., press any key).

Import Function:

1. Open the file "input.csv" for reading.
2. Create an empty itemQueue.
3. Read items from the file and add them to the itemQueue.
4. Close the file.
5. Display import status.
6. Return the itemQueue.

Add Function:

1. Get item details (ID, name, price, location) from the user.
2. Create a new goods object with the details provided.
3. Return the goods object.

printHistory Function:

1. Open the file "history.txt" for writing.
2. Iterate through the historyStack:
 - a. Write each history node's details (ID, name, price, location, action) to the file.
3. Close the file.

saveToFile function:

1. Open file "output.csv"
2. While item is not empty
 - a. Write item info into file
 - b. Item.dequeue

3. Close file

6.0 Description of how data structure operations

The provided C++ code for the Warehouse Inventory Management System utilizes two main data structures: a queue and a stack. Below is an explanation of how these data structures are employed in the code:

6.1 Queue Operations:

The queue has been used to manage inventory items, and it follows the First In, First Out (FIFO) principle.

Queue Declaration:

```
class itemNodeQueue{
public:
    goods item;
    itemNodeQueue *next;
    itemNodeQueue *prev;
};

class itemQueue{
public:
    itemNodeQueue *front;
    itemNodeQueue *rear;
    int size;

    void createQueue();
    bool isEmpty();
    void enqueue(goods item);
    void dequeue();
    goods getItem();
    void display();
};
```

Queue Initialization:

```
void itemQueue::createQueue(){
    front = NULL;
    rear = NULL;
    size = 0;
}
```

Initialize the front, rear pointers, and sets the size to zero.

Check if Queue is Empty:

```

bool itemQueue::isEmpty(){
    if(front == NULL)
        return true;
    else
        return false;
}

```

Return true if the queue is empty; otherwise, returns false.

Enqueue Operation:

```

void itemQueue::enqueue(goods item){
    itemNodeQueue *newNode = new itemNodeQueue;
    newNode->item = item;
    newNode->next = NULL;
    newNode->prev = NULL;

    if(isEmpty()){
        front = newNode;
        rear = newNode;
    }else{
        rear->next = newNode;
        newNode->prev = rear;
        rear = newNode;
    }
    size++;
}

```

Add new inventory item to the end of the queue.

Dequeue Operation:

```

void itemQueue::dequeue(){
    if(isEmpty()){
        cout << "Queue is empty" << endl;
    }else{
        itemNodeQueue *temp = front;
        front = front->next;
        if(front == NULL)
            rear = NULL;
        else
            front->prev = NULL;
        delete temp;
        size--;
    }
}

```

Remove the front inventory item from the queue.

Get Front Item:

```

goods itemQueue::getItem(){
    return front->item;
}

```

Return the item in front of the queue.

Display Queue:

```

void itemQueue::display(){
    itemNodeQueue *temp = front;
    cout << "ID\tName\tPrice\tLocation" << endl;
    while(temp != NULL){
        cout << temp->item.getId() << "\t" << temp->item.getName() << "\t\t" << temp->item.getPrice() << "\t\t" << temp->item.getItemLocation() << endl;
        temp = temp->next;
    }
}

```

Displays all items in the queue.

6.2 Stack Operations:

The stack is used to track the history of inventory actions.

Stack Declaration:

```

class historyNodeStack{
public:
    goods item;
    char action;
    historyNodeStack *next;
};

class historyStack{
private:
    historyNodeStack *top;
    historyNodeStack *bottom;
    int size;

public:
    void createStack();
    bool isEmpty();
    void push(goods item, char action);
    void pop();
    char getAction();
    goods getItem();
};

```

Stack Initialization:

```
void historyStack::createStack(){
    top = NULL;
    bottom = NULL;
    size = 0;
}
```

Initialize the top and bottom pointers and set the size to zero.

Check if Stack is Empty:

```
bool historyStack::isEmpty(){
    if(top == NULL)
        return true;
    else
        return false;
}
```

Return true if the stack is empty; otherwise, it returns false.

Push Operation:

```
void historyStack::push(goods item, char action){
    historyNodeStack *newNode = new historyNodeStack;
    newNode->item = item;
    newNode->action = action;
    newNode->next = NULL;

    if(isEmpty()){
        top = newNode;
        bottom = newNode;
    }else{
        top->next = newNode;
        top = newNode;
    }
    size++;
}
```

Add new inventory items and action to the top of the stack.

Pop Operation:

```

void historyStack::pop(){
    if(isEmpty()){
        cout << "Stack is empty" << endl;
    }else{
        historyNodeStack *temp = bottom;
        historyNodeStack *prev = NULL;
        while(temp->next != NULL){
            prev = temp;
            temp = temp->next;
        }
        if(prev == NULL){
            top = NULL;
            bottom = NULL;
        }else{
            prev->next = NULL;
            top = prev;
        }
        delete temp;
        size--;
    }
}

```

Remove the item from the top of the stack.

Get Top Action:

```

char historyStack::getAction(){
    if(isEmpty())
        cout << "Stack is empty" << endl;
    else
        return top->action;
}

```

Return the action associated with the top item in the stack.

Get Top Item:

```

goods historyStack::getItem(){
    if(isEmpty())
        cout << "Stack is empty" << endl;
    else
        return top->item;
}

```

Return the item to the top of the stack.

6.3 Additional Operations:

Import Items from File:

```
itemQueue import(){
    ifstream file("input.csv");
    int id;
    string name;
    double price;
    string itemLocation;
    itemQueue item;
    item.createQueue();
    goods *newItem;
    system("cls");
    displayHeader();
    if (!file.is_open()) {
        cout << "File not found" << endl;
    }else{
        cout << "Importing..." << endl;
        //each line is a goods
        while (!file.eof()) {
            file >> id;
            file.ignore();
            getline(file, name, ',');
            file >> price;
            file.ignore();
            getline(file, itemLocation);
            newItem = new goods(id, name, price, itemLocation);
            item.enqueue(*newItem);
        }
    }
    file.close();
    cout << "Import successful!" << endl;
    system("pause");
    return item;
}
```

Read inventory items from a file and enqueues them into the item queue.

Add New Item:

```
goods *add(){
    int id;
    string name;
    double price;
    string itemLocation;
    goods *newItem;
    system("cls");
    displayHeader();
    string input;
    while(true){
        cout << "Enter item ID: ";
        getline(cin, input);
        if(isNumber(input)){
            id = stoi(input);
            break;
        }else{
            cout << "Invalid input, please enter a number!" << endl;
            system("pause");
            system("cls");
            displayHeader();
        }
    }
}
```



```

    cout << "Enter item name: ";
    getline(cin, name);
    while(true){
        cout << "Enter item price: ";
        getline(cin, input);
        if(isfloat(input)){
            price = stod(input);
            break;
        }else{
            cout << "Invalid input, please enter a number!" << endl;
            system("pause");
            system("cls");
            displayHeader();
        }
    }
    cout << "Enter item location: ";
    getline(cin, itemLocation);
    newItem = new goods(id, name, price, itemLocation);
    return newItem;
}

```

Prompt the user to input details for a new item and returns a dynamically allocated goods object representing the new item.

Print History to File:

```

void printHistory(historyStack history){
    ofstream file("history.txt");
    if(file.is_open()){
        file <<setw(5)<< "ID" <<setw(15)<< "Name" <<setw(15)<< "Price" <<setw(15)<< "Location" <<setw(15)<< "Action" << endl;
        while(!history.isEmpty()){
            file <<setw(5)<< history.getItem().getId() <<setw(15)<< history.getItem().getName() <<setw(15)<< history.getItem().getPrice()
            <<setw(15)<< history.getItem().getItemLocation() <<setw(15)<< history.getAction() << endl;
            history.pop();
        }
    }
    file.close();
}

```

Write the history stack contents to a file (history.txt).

Save Inventory to File:

```

void saveToFile(itemQueue item){
    ofstream file("output.csv");
    if(file.is_open()){
        while(!item.isEmpty()){
            file << item.getItem().getId() << "," << item.getItem().getName() << "," << item.getItem().getPrice() << "," << item.getItem().getItemLocation() << endl;
            item.dequeue();
        }
    }
    file.close();
}

```

Write the inventory items to a file (output.csv).

7.0 User manual or guide

This program's primary purpose is to help manage stockroom inventory.

The function of the program is as below

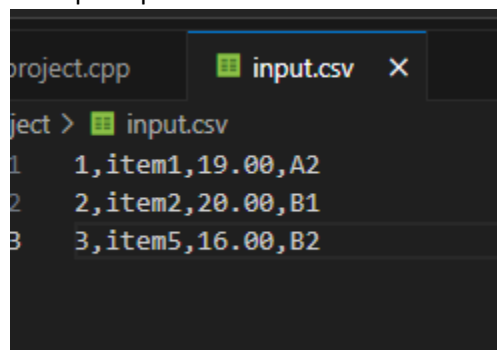
In the main menu, user can perform the following tasks:

```
=====
|           Warehouse Inventory Management           |
|                               System                               |
|=====|
Please select an option:
1. Import item from file
2. Add item
3. Remove item
4. Display item
5. Print history, save files, and exit
Enter your choice: █
```

1. Import items from file:

User can directly import a list of items from the file named "input.csv":

Example input.csv file:



The screenshot shows a code editor with two tabs: 'project.cpp' and 'input.csv'. The 'input.csv' tab is active, displaying the following content:

```
project > input.csv
1 1,item1,19.00,A2
2 2,item2,20.00,B1
3 3,item5,16.00,B2
```

Output after importing items:

```
=====
|   Warehouse Inventory Management   |
|                   System            |
=====
Importing...
Import successful!
Press any key to continue . . .
```

2. Add item manually:

In addition, user can also add the items manually into the system:

```
=====
|   Warehouse Inventory Management   |
|                   System            |
=====
Enter item ID: 4
Enter item name: item20
Enter item price: 19.80
Enter item location: A4
Item added!
Press any key to continue . . .
```

- 3.
4. Remove item:

First in first out (FIFO) concept ensures that the oldest items in the inventory are the first to be used or shipped. This helps in rotating stock efficiently, preventing perishable goods expiring or becoming obsolete. It's particularly important for industries where products have a limited shelf life.

```
=====
|   Warehouse Inventory Management   |
|                   System            |
=====
Item: item1 id: 1 has been taken out!
Press any key to continue . . .
```

5. Display all items in the queue of the system

```
=====
|      Warehouse Inventory Management      |
|                      System              |
=====
ID      Name      Price      Location
1      item1      19         A2
2      item2      20         B1
3      item5      16         B2
Press any key to continue . . .
```

6. Save and exit:

This selection will save the items into output.csv files, and history into history.txt files and exit the system.

The program display:


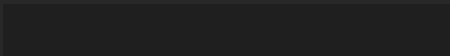
```
=====
|      Warehouse Inventory Management      |
|                      System              |
=====
Printing history...
History saved into file
Saving inventory...
Inventory saved into file
Press any key to continue . . .
```

The history.txt file:

```
project > history.txt
1  ID      Name      Price      Location      Action
2  21      A thing      20         G1            a
3  1       item1      19         A2            r
4
```

The actions are represented by a character, which is a for add, and r for removing.

The output.csv file:

```
object >  output.csv  
1      2,item2,20,B1  
2      3,item5,16,B2  
3      21,A thing,20,G1  
4      
```