PROJECT REPORT

NAME: Jason Tan Thong Shen STUDENT ID: 20060534

A. Information for Use

a. Introduction:

My program functions exactly like the assignment requires it. Most of the functionality such as Loading data are loading location file, inventory file, product file, order file, serialized file, and then displaying the location overview, inventory overview, do product search, find and display inventory for a location, find and display distance between two locations, find and display route for collecting order, save into serialized data and exit program are available in the program.

b. Installation:

i. **To compile:** javac *.java
ii. **droneCollector.java** requires DSAGraph.java and DSALinkedList.java
iii. **DSAGraph.java** requires DSALinkedList.java to function
iv. **DSALinkedList.java** requires DSAQueue.java, DSAStack.java, and DSAGraphVertex.java to work properly for the DSAGraph.java
v. **DSAGraphVertex.java** requires DSALinkedList.java to allow the DSAGraph.java to work and store vertices.
vi. **(ii.) to (v.) depend on each other**
vii. **order90.csv** is a new order file that I created to collect orders and sort them into collected items and uncollected items.

c. Terminology/Abbreviations:

1. **oneAbove** represents **Qty1+**
2. **tenAbove** represents **Qty10+**
3. **twtyAbove** represents **Qty25+**

d. Walkthrough:

<u>Load data:</u>

It asks user to enter a **location** file, the system will read it and **store** the location data such as edges and distances between two locations into the program.
No error message will be printed if there are no errors reading the file.
An error message will be printed if user enters a wrong file or file with the wrong format.

<u>Inventory data:</u>

It asks user to enter an **Inventory** file, the system will read it and **store** the inventory data such as product names and stocks on hand into the program.
No error message will be printed if there are no errors reading the file.
An error message will be printed if user enters a wrong file or file with the wrong format.

Product data:

It asks user to enter a **product** file, the system will read it and **store** the product data such as product name, description, Qty1+, Qty10+, and Qty25+ into the program.
No error message will be printed if there are no errors reading the file.
An error message will be printed if user enters a wrong file or file with the wrong format.

Order data:

It asks user to enter an **order** file, the system will read it and **store** the order data such as the date, contact, address, product name, and number of items to collect for each product into the program.
No error message will be printed if there are no errors reading the file.
An error message will be printed if user enters a wrong file or file with the wrong format.

Serialized data:

The system automatically loads the **serialized** file into the program and **restore** all the data as it was saved.
No error message will be printed if there are no errors reading the file.
An error message will be printed if the serialized file in outdated. You need to save it again, through using the program and saving it.

Location overview:

It will display the locations in the graph, number of vertices, number of edges, and display each location with its edges and edge distances.
If no data to display, it will ask user to load the location data first.

Inventory overview:

It will display the inventories of each locations with the total number of products listed below every location's inventory section.
If no data to display, it will ask user to load the inventory data first.

Product search:

It will ask user to enter a product name to search, the system will search the product, gather the product details such as product name, description, Qty1+, Qty10+, Qty25+ and display it for the user to see.
If no data to display, it will ask user to load the product data first.
If product doesn't exist in the product file, it will tell user that the entered product doesn't exist within the system and product file.

Find and display inventory for a location:

Ask user for a location, and displays the inventory with stocks on hand of that specific location only.
If no data to display, it will ask user to load the inventory file.
If location doesn't exist, it will display an error and ask user to try another location.

Find and display distance between two locations:

Ask user for two locations, and displays the "STARTPOINT", "DESTINATION", "TOTAL DISTANCE" to the user.
If no data to display it will ask user to load location file.
If user didn't enter two locations, it will notify the user to make sure user enters two locations.

Find and display route for collecting order:

Gather the order file data and display the route it travelled through Breath First Search, it displays the items collected and at which location. It also displays any uncollected items. It uses order90.csv file to test the uncollected items scenario.
If no data to display, it asks user to load the order file.

Save data (serialized):

It saves the data into a file called "serialized.txt", it saves the DSAGraph class and anything else that is linked with DSAGraph class.
It never gives any error as it just saves data.

Exit:

It will terminate the program and exit. It never gives any error. If it does, you won't be able to exit the program, which is highly unlikely that this error will happen.

e. Future work:

I handled all the **error handling** using try-catch.
Enhanced formatting of displayed information to help users analyze my program better.
Neat program code for markers to mark with **very detailed explanations** on all methods available to avoid confusion.

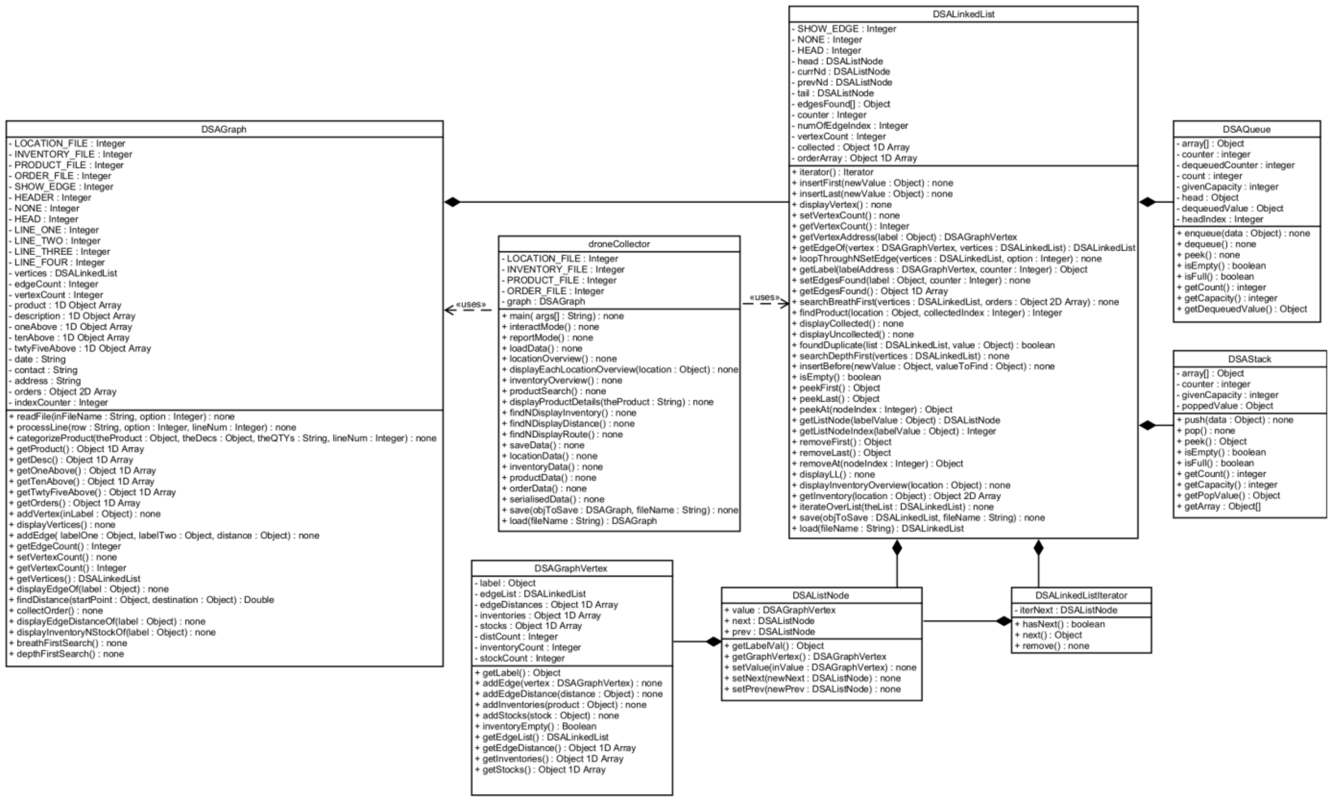
B. Traceability Matrix

[Note: "(…)" means it contains arguments, it is too long to include in the traceability matrix column]
"droneCollector.reportMode(…)" means you can find the "reportMode" method in the "droneCollector" class
Note: "Test" section without a testing method requires you to run the program and test it manually and visually. IF there are test methods available, you can run "UnitTestDSAGraph.java" and get results immediately.

		Requirements	**Design/Code**	**Test**
1	**Modes**	System checks for mode "-i" or "-r" or no arguments	droneCollector.main(…)	Test by running the program with "-i" or "-r"
2	**Report mode**	Receives the files from command line and display them onto terminal. Command: "droneCollector -r stores.csv catalogue.csv inventories.csv order90.csv"	droneCollector.reportMode(…)	Test by running the program with correct files (report mode)
3	**Interactive mode**	Ask for user input to enter an option between **4** to **11 located <u>below in this traceability matrix</u>**	droneCollector.interactMode()	Test by running the program with "-i", you will see the menu and it asks for user input.
4	**Load Data**	Ask for user input to enter an option between **4.1** and **4.5 located <u>below in this traceability matric under this "Load Data" section.</u>**	droneCollector.loadData()	Test by running the program with "-i", enter an option to go to the "Load Data" section, and you will see this running.
4.1	**(Location Data)**	Ask user for their **location file** and system reads it	droneCollector.locationData()	Go to this section, give it your file and observe for any reading file error. If not, test successful. [OR] **UnitTestDSAGraph.testReadLocation()**
4.2	**(Inventory Data)**	Ask user for their **inventory file** and system reads it	droneCollector.inventoryData()	Go to this section, give it your file and observe for any reading file error. If not, test successful. [OR] **UnitTestDSAGraph.testReadInventory()**
4.3	**(Product Data)**	Ask user for their **product file** and system reads it	droneCollector.productData()	Go to this section, give it your file and observe for any reading file error. If not, test successful. [OR]

				UnitTestDSAGraph.testReadProduct()
4.4	**(Order Data)**	Ask user for their **order file** and system reads it	droneCollector.orderData()	Go to this section, give it your file and observe for any reading file error. If not, test successful. [OR] **UnitTestDSAGraph.testReadOrder()**
4.5	**(Serialized Data)**	Ask user for their **serialized file** and system reads it	droneCollector.serialisedData()	Go to this section, give it your file and observe for any reading file error. If not, test successful. [OR] **UnitTestDSAGraph.testReadSerialized()**
5	**Location Overview**	**Acquire each location** and **display the location and its edges**	droneCollector.locationOverview()	Go to this section, check all the displayed information. If no error, test successful.
6	**Inventory Overview**	**Display** <u>each</u> location's inventory	droneCollector.inventoryOverview()	Go to this section, check all the displayed information. If no error, test successful.
7	**Product Search**	**Ask user for a product** and **displays the product information**	droneCollector.productSearch()	Go to this section, insert product, check displayed information. If no errors, test successful.
8	**Find and Display inventory for a location (store)**	**Ask user for location** and **displays the location's inventory**	droneCollector.findNDisplayInventory()	Go to this section, insert location, check displayed inventories, if no error, test successful.
9	**Find and Display Distance between two location (store)**	**Ask user for two locations** and **displays the distance**	droneCollector.findNDisplayDistance()	Go to this section, insert 2 locations, check displayed inventories, if no error, test successful.
10	**Find and Display route for collecting order**	**Get data from the submitted order file** and display the route and at which location the item was collected, and also display uncollected items	droneCollector.findNDisplayRoute()	Go to this section, and check displayed information, if no error, test successful.
11	**Save data (serialized)**	Saves the whole program in a **serialize format** into a file	droneCollector.saveData()	Go to this section, and check if any saving error, if not test successful.

C. **CLASS DIAGRAM** (you may need to zoom in to see my classes)



```
DSALinkedList
- SHOW_EDGE : Integer
- NONE : Integer
- HEAD : Integer
- head : DSAListNode
- currNd : DSAListNode
- prevNd : DSAListNode
- tail : DSAListNode
- edgesFound[] : Object
- counter : Integer
- numOfEdgeIndex : Integer
- vertexCount : Integer
- collected : Object 1D Array
- orderArray : Object 1D Array
+ iterator() : Iterator
+ insertFirst(newValue : Object) : none
+ insertLast(newValue : Object) : none
+ displayVertex() : none
+ setVertexCount() : none
+ getVertexCount() : Integer
+ getVertexAddress(label : Object) : DSAGraphVertex
+ getEdgeOf(vertex : DSAGraphVertex, vertices : DSALinkedList) : DSALinkedList
+ loopThroughNSetEdge(vertices : DSALinkedList, option : Integer) : none
+ getLabel(labelAddress : DSAGraphVertex, counter : Integer) : Object
+ setEdgesFound(label : Object, counter : Integer) : none
+ getEdgesFound() : Object 1D Array
+ searchBreathFirst(vertices : DSALinkedList, orders : Object 2D Array) : none
+ findProduct(location : Object, collectedIndex : Integer) : Integer
+ displayCollected() : none
+ displayUncollected() : none
+ foundDuplicate(list : DSALinkedList, value : Object) : boolean
+ searchDepthFirst(vertices : DSALinkedList) : none
+ insertBefore(newValue : Object, valueToFind : Object) : none
+ isEmpty() : boolean
+ peekFirst() : Object
+ peekLast() : Object
+ peekAt(nodeIndex : Integer) : Object
+ getListNode(labelValue : Object) : DSAListNode
+ getListNodeIndex(labelValue : Object) : Integer
+ removeFirst() : Object
+ removeLast() : Object
+ removeAt(nodeIndex : Integer) : Object
+ displayLL() : none
+ displayInventoryOverview(location : Object) : none
+ getInventory(location : Object) : Object 2D Array
+ iterateOverList(theList : DSALinkedList) : none
+ save(objToSave : DSALinkedList, fileName : String) : none
+ load(fileName : String) : DSALinkedList
```

```
DSAGraph
- LOCATION_FILE : Integer
- INVENTORY_FILE : Integer
- PRODUCT_FILE : Integer
- ORDER_FILE : Integer
- SHOW_EDGE : Integer
- HEADER : Integer
- NONE : Integer
- HEAD : Integer
- LINE_ONE : Integer
- LINE_TWO : Integer
- LINE_THREE : Integer
- LINE_FOUR : Integer
- vertices : DSALinkedList
- edgeCount : Integer
- vertexCount : Integer
- product : 1D Object Array
- description : 1D Object Array
- oneAbove : 1D Object Array
- tenAbove : 1D Object Array
- twtyFiveAbove : 1D Object Array
- date : String
- contact : String
- address : String
- orders : Object 2D Array
- indexCounter : Integer
+ readFile(inFileName : String, option : Integer) : none
+ processLine(row : String, option : Integer, lineNum : Integer) : none
+ categorizeProduct(theProduct : Object, theDecs : Object, theQTYs : String, lineNum : Integer) : none
+ getProduct() : Object 1D Array
+ getDesc() : Object 1D Array
+ getOneAbove() : Object 1D Array
+ getTenAbove() : Object 1D Array
+ getTwtyFiveAbove() : Object 1D Array
+ getOrders() : Object 1D Array
+ addVertex(inLabel : Object) : none
+ displayVertices() : none
+ addEdge( labelOne : Object, labelTwo : Object, distance : Object) : none
+ getEdgeCount() : Integer
+ setVertexCount() : none
+ getVertexCount() : Integer
+ getVertices() : DSALinkedList
+ displayEdgeOf(label : Object) : none
+ findDistance(startPoint : Object, destination : Object) : Double
+ collectOrder() : none
+ displayEdgeDistanceOf(label : Object) : none
+ displayInventoryNStockOf(label : Object) : none
+ breathFirstSearch() : none
+ depthFirstSearch() : none
```

```
droneCollector
- LOCATION_FILE : Integer
- INVENTORY_FILE : Integer
- PRODUCT_FILE : Integer
- ORDER_FILE : Integer
- graph : DSAGraph
+ main( args[] : String) : none
+ interactMode() : none
+ reportMode() : none
+ loadData() : none
+ locationOverview() : none
+ displayEachLocationOverview(location : Object) : none
+ inventoryOverview() : none
+ productSearch() : none
+ displayProductDetails(theProduct : String) : none
+ findNDisplayInventory() : none
+ findNDisplayDistance() : none
+ findNDisplayRoute() : none
+ saveData() : none
+ locationData() : none
+ inventoryData() : none
+ productData() : none
+ orderData() : none
+ serialisedData() : none
+ save(objToSave : DSAGraph, fileName : String) : none
+ load(fileName : String) : DSAGraph
```

```
DSAQueue
- array[] : Object
- counter : integer
- dequeuedCounter : integer
- count : integer
- givenCapacity : integer
- head : Object
- dequeuedValue : Object
- headIndex : Integer
+ enqueue(data : Object) : none
+ dequeue() : none
+ peek() : none
+ isEmpty() : boolean
+ isFull() : boolean
+ getCount() : integer
+ getCapacity() : integer
+ getDequeuedValue() : Object
```

```
DSAStack
- array[] : Object
- counter : integer
- givenCapacity : integer
- poppedValue : Object
+ push(data : Object) : none
+ pop() : none
+ peek() : Object
+ isEmpty() : boolean
+ isFull() : boolean
+ getCount() : integer
+ getCapacity() : integer
+ getPopValue() : Object
+ getArray : Object[]
```

```
DSAGraphVertex
- label : Object
- edgeList : DSALinkedList
- edgeDistances : Object 1D Array
- inventories : Object 1D Array
- stocks : Object 1D Array
- distCount : Integer
- inventoryCount : Integer
- stockCount : Integer
+ getLabel() : Object
+ addEdge(vertex : DSAGraphVertex) : none
+ addEdgeDistance(distance : Object) : none
+ addInventories(product : Object) : none
+ addStocks(stock : Object) : none
+ inventoryEmpty() : Boolean
+ getEdgeList() : DSALinkedList
+ getEdgeDistance() : Object 1D Array
+ getInventories() : Object 1D Array
+ getStocks() : Object 1D Array
```

```
DSAListNode
+ value : DSAGraphVertex
+ next : DSAListNode
+ prev : DSAListNode
+ getLabelVal() : Object
+ getGraphVertex() : DSAGraphVertex
+ setValue(inValue : DSAGraphVertex) : none
+ setNext(newNext : DSAListNode) : none
+ setPrev(newPrev : DSAListNode) : none
```

```
DSALinkedListIterator
- iterNext : DSAListNode
+ hasNext() : boolean
+ next() : Object
+ remove() : none
```

D. **Class Descriptions**

a. **droneCollector.java**

Description:

This class is the main functionality handler and administrator for the entire program, it handles the front-end menu user interfaces as well as directing user inputs to other classes (back-end code) to perform loading data, storing data, and sorting data.
This class handles user input errors and provides useful error instruction to help user navigate through the program. It performs all functionality stated in section 2 of the assignment including report and interactive modes. After rigorous testing, there isn't any bugs spotted which meant the error handling is doing its job well and the software is working fine as it should.

b. **DSAGraph.java**

Description:

This is the core program code that stores the state of the program data. The serialize file highly depends on this class to be able to save and restore the state of the program. This class makes constructing the graph possible, it also helps droneCollector.java to load, store, categorize, calculate, and search data that relates to the graph all in one convenient class.

c. **DSALinkedList.java (public class) & DSAListNode.java (private inner class)**
Description:
This class is an extension of DSAGraph.java. This class handles the Linked List and List Node that constructs the Graph. DSALinkedList class helps DSAGraph class to get the address of a specific list node address or DSAGraphVertex address. It stores each location into the list node that will redirect to DSAGraphVertex for the rest of the data storage for each location. DSAListNode class acts as a connector between DSALinkedList class and DSAGraphVertex class. With this setup, DSAGraph class can call DSALinkedList class to find out the location's DSAGraphVertex class and after that communicate with DSAGraphVertex.
This class also has a few methods to collect and find the route of the order. It also contains breath first search to travel through the linked list and edges to find the locations that has the order.
This class also helps display collected items, uncollected items, and display inventory overviews.

d. **DSAListNode.java**
Description:
This class stores the location into DSAGraphVertex class it also sets the next and previous list node and also has a useful function to retrieve location value back to caller.

e. **DSAGraphvertex.java**
Description:
This class is important to store the data of each location such as location's name, location's edgeList, location's edgeDistances, location's inventories and stocks, number of stocks, number of inventories, and returns the data back to caller if caller from other classes needs these data from this class. This is created to separate the linked list, graph, and the vertices.

f. **DSALinkedListIterator.java**
Description:
Its main purpose is to iterate through the linked list to display each location available in the linked list.

g. **DSAQueue.java**
Description:
This class helps the breath first search in the DSALinkedList class to function by using the enqueue and dequeue.
I created this because it is easier to manage so I know where the queue code is.

h. **DSAStack.java**
Description:
This class helps the depth first search in the DSALinkedList class to function by using the push and pop.

I created this because it is easier to manage so I know where the stack code is. Suppose to be used for the multiple hops to find the location between two locations but time was limited.

E. Justification

Reason why I chose Breath First Search to collect order:
I want to collect the order from the nearest stores/locations first before I collect the order from further locations. If the nearest locations don't have the order items, then the program will go to the further locations to look for the order items. Uncollected items will not be collected if not found in all locations.

Reason why I chose arrays for variables in DSAGraphVertex class instead of using LinkedList:
I understand that Linked list is a better option when we don't know exactly how many data is going to be stored. However, the linked list in this program is specially built to handle the edgesList from DSAGraphVertex and graph locations from DSALinkedList and DSAGraph, not to store the variables located for DSAGraphVertex class.
You may refer to "Extra information for visual aid" section to understand the flow of my program.

Question about the order file format:
Due to late announcement, the valid order files that the program can accept are the order files from the small set, not the medium set.

Reason why there are so many private class fields in DSAGraph class:
DSAGraph is a critical program code that will be stored into a serialized file and restore the program's state back to where it was last saved when needed. Therefore, most class fields are located in DSAGraph class.

Why is DSALinkedList class so long:
There are some methods that are from the practical that is not used for the assignment, I built the assignment on it by adding extra methods, I didn't erase the methods that I'm not using for this assignment.

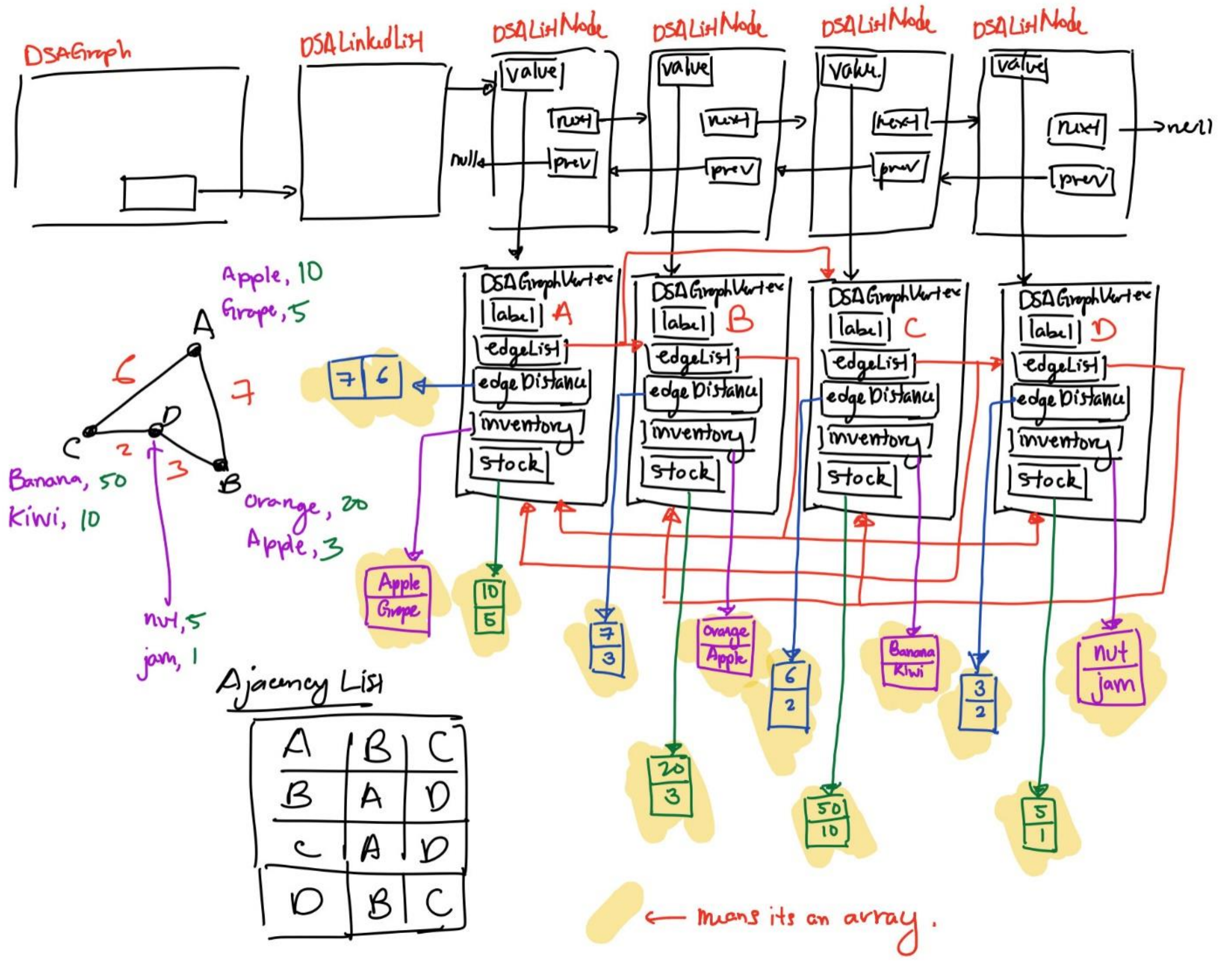
Reason why DSAListNode class is a private inner class:
To avoid creating too many classes, I tried to put it in the DSALinkedList.java file since it is related to one another.

Reason why there is a file called "order90.csv":
This order file is created by myself to test the collected items and uncollected items test cases. Some products in the order file does not exist in the inventories of all locations which will remain uncollected while other products available in the inventories will be collected.

F. Reference

none (everything is referred from my DSA practical labs 2020, and my DSA practical labs 2021, and DSA assignment requirements 2021, no online sources)

Extra information for visual aid:



Ajacency List

A	B	C
B	A	D
C	A	D
D	B	C