BER2013: ALGORITHM DESIGN AND ANALYSIS

ACADEMIC SESSION: MAY-AUG 2023

Project Name: Group 2
Group Member:

1) Lim Chee Yang	1002059327
2) Ali Isam Husam Al-Turaihi	1002164751
3) Mohamed Tarek Essam Hessein Emad	1002163249
4) Lim Jiun Hong	1002164511
5) Chong Wen Qi	1002058146

INSTRUCTIONS TO CANDIDATES

1. This assignment will contribute **30%** to your final grade.
2. This coursework is a group assignment. You may have maximum 5 members in one group.

IMPORTANT

Plagiarism is not allowed and any student found to commit such an act, 0 mark will be awarded.

Courseworks must be submitted on their due dates. If a coursework is submitted after its due date, the following penalty will be imposed:

- **ONE day late : 3 marks deducted from the total marks awarded.**
- **TWO days : 5 marks deducted from the total marks awarded**
- **THREE or more days : Assignment will not be marked and 0% will be awarded.**

For example: *A student scores 10 marks for an assignment that has a total mark of 10. If the assignment is submitted one day late, the marks awarded will be 7 marks.*

BER2013 Algorithm and Analysis
May - Aug 2023
Assignment Rubric

Final report:

	Unsatisfactory (0 to 4 marks)	**Satisfactory (5 to 6marks)**	**Good (7 to 8marks)**	**Excellent (9 to 10 marks)**	**Marks Scored**
Delivery PLO 2	• Completed less than 70% of the requirements. • Not delivered on time or not in correct format	• Completed between 70-80% of the requirements. • Delivered on time, and in correct	• Completed between 80-90% of the requirements. • Delivered on time, and in correct format	• Completed between 90-100% of the requirements. • Delivered on time, and in correct format	
Documentation /Algorithm PLO2	• No or incomplete documentation included.	• Basic documentation has been completed including descriptions of all variables. • Purpose is noted for each method. • Little to non-discussion and analysis is provided	• Clearly documented including descriptions of all variables. • Specific purpose is noted for each algorithm and data structure. • Acceptable discussion and analysis is provided	• Clearly and effectively documented including descriptions of all variables. • Specific purpose is noted for each method, algorithm, data structure, input requirements, and output results. • discussion and analysis is provided	
				TOTAL	

Codes:

Coding Standards PLO 5	• Poor use of white space (indentation, blank lines). • Disorganized and messy • Poor use of class, variables and method (ambiguous naming).	• White space makes program easy to read. • Organized work. • Good use of class, variables and method (unambiguous naming).	• Good use of white space. • Organized work. • Good use of class, variables and method (unambiguous naming)	• Excellent use of white space. • Creatively organized work. • Excellent use of class, variables and method (Unambiguous naming).	

Use of Equipment / Tool **PLO 5**	• Student unable to use tools	• Uses tools, equipment and materials with limited/some competence	• Uses tools, equipment and materials with considerable competence	• Uses tools, equipment and materials with a high degree of competence	
				Total	

Presentation

	Unsatisfactory (0 to 2 marks)	Satisfactory (3 marks)	Good (4 marks)	Excellent (5 marks)	Marks Scored
Comprehension **PLO 2**	• Presenters did not understand the topic. • Majority of questions answered by only one member or majority of information incorrect.	• Few members showed good understanding of some parts of the topic. • Only some members accurately answered questions.	• Most showed a good understanding of the topic. • All members able to answer most of audience questions.	1. Extensive knowledge of the topic. 2. Members showed complete understanding of assignment. Accurately answered all questions posed.	
Presentation Skills **PLO2**	• Minimal eye contact by more than one member focusing on small part of audience. • The audience was not engaged. • Majority of presenters spoke too quickly or quietly making it difficult to understand. • Inappropriate/disinterested body language.	• Members focused on only part of audience. • Sporadic eye contact by more than one presenter. The • audience was distracted. • Speakers could be heard by only half of the • audience. Body language was distracting.	• Most members spoke to majority of audience; steady eye • contact. The audience was engaged by the • presentation. Majority of presenters spoke • at a suitable volume. Some fidgeting by member(s).	• Regular/constant eye contact, the audience was engaged, and presenters held the audience's • attention. Appropriate speaking volume & body language.	
Contents **PLO2**	• The presentation was a brief look at the topic but many questions were left unanswered. • Majority of information irrelevant and significant points left out.	• The presentation was informative but several elements went • unanswered. Much of the information irrelevant; coverage of some of major points.	• The presentation was a good summary of the topic. • Most important information covered; little irrelevant info.	• The presentation was a concise summary of the topic with all • questions answered. Comprehensive and complete coverage of information.	
				Total	

Table of Contents

CHAPTER I: INTRODUCTION

BACKGROUND STUDY

Java is a popular programming language used for developing enterprise applications, making it a suitable choice for building a supermarket inventory management system. Java is known for its platform independence, which means that the system can run on any platform that supports Java. This makes it easier to deploy the system across different devices and operating systems. Additionally, Java has a large and active developer community, which means that there are many libraries and frameworks available for Java developers to use. This can help to speed up development time and reduce errors. Finally, Java is known for its reliability and security, which is essential for a system that manages sensitive information such as library records and user data.

In today's fast-paced world, efficient supermarket inventory management is critical to smooth operations and meeting customer needs. We are building a software program for supermarket operators to streamline the management process and increase efficiency. This program seeks to make many jobs easier, such as adding, removing, reading, sorting, searching, and keeping an inventory database up to date. Our program will give flexibility and efficiency by incorporating various data structures and algorithms to suit the requirements of various supermarkets.

The supermarket business is constantly flooded with consumers, demanding excellent inventory management systems to enable prompt replenishment and precise product monitoring. Traditional manual inventory management methods frequently result in mistakes, time-consuming activities, and probable stockouts. To overcome these issues, our software employs innovative technology to automate and optimize supermarket management procedures.

To summarize, our supermarket management software provides an innovative solution for performing inventory-related activities efficiently. We hope to improve the overall user experience for supermarket owners by using various data formats and algorithms. The user-friendly design and powerful functionality of the program will help to streamline processes and maximize client happiness.

1.2 PROBLEM STATEMENT

One of the most common frustrations faced by supermarket operators or owners is inaccurate stock level. Human tends to make mistake during work due to many factors such as lack of sleeping, distraction, improper working environment, and other factors. The same thing happened in the operation of supermarket too, the traditional way of stock counting has caused various unnecessary mistakes, for instance, manual stock counting and recording can lead to inaccuracies in stock levels, causing inventory discrepancies and potentially leading to stockouts or overstocking of products which will lead to severe consequences and losses to the supermarket.

In addition, inefficient stock tracking is also one of the problems that supermarket's owner trying to solve in real life. Stock tracking is a relatively time consuming and human power relying on job scope among others jobs available in supermarkets. For example, there are over hundreds type of things we could get from a typical neighbourhood supermarket, considering different brands from the same type of product, it can get over thousandths or more than that of inventory to track. Supermarket's staff have to spend a significant amount of time and effort in tracking every stock in the supermarket and make sure the result is accurate and tally with the supermarket's inventory stock count list.

Besides, supermarket staff and owner frequently struggling with the exact price listed for each and every inventory available in the supermarket. Finding and tagging different prices for different types and brands of products could be burdensome and crucial especially there are over thousandths of inventory to handle in a supermarket. Without an efficient inventory management system, the staffs would need to manually check the price of products with their supervisor or manager which will reduce the efficiency of the supermarket's operation yet lower the reputation of the supermarket when they are unsure about the price if a customer approach them to enquire regarding the product's related information.

Last but not least, the supermarket could not make an analysis report on the types and brands of inventory they should prioritize and readjust on the following restock procedure. With the aid of an inventory management system, the owner could keep an eye on the changes of the amount of inventory on a weekly or monthly basis to conclude that which product is the best-selling product and which product is not interested by the consumers. By analysing in this way, it could help to enhance and boost the sales of the supermarket in a long run, as consumers will get attracted to the product they are looking for with a reasonable price.

1.3 OBJECTIVE

The objective of making a supermarket inventory management system is to enhance and optimize the management of supermarket inventory efficiently and accurately. This system aims to automate various inventory-related tasks, provide real-time insights into inventory quantity and prices, and improve overall operational efficiency. Tha main features designed in our program is to add, remove, search, and sort the supermarket's inventory easily and efficiently through a management system.

In an operating supermarket, the inventory would be adjusted repeatedly according to the latest trend and also consumer's preference, thus add and remove features are included in our project system. This could help the staff and owner of the supermarket to amend the inventory list and keep the list updated by just adding and removing the inventory into and from the list respectively easily. This inventory management system helps to ease the job of the supermarket's staff and owner in making themselves familiar with every product available in the supermarket by just checking the system and updating the product in the supermarket by just clicking on the system without any extra efforts required.

Additionally, this management systems could help in enhancing the efficiency of the cashier while processing and checking out the products purchased by customers at counter. For instance, the cashier would just require using the search() function in the program to obtain the price of a particular product by typing the name or ID of the product, obtaining the desired information from the system easily. With the existence of the inventory management system, the cashier could ensure that the total amount of the inventory purchased by consumers is correct, hence reducing the conflict occurs between consumers and cashier due to technical issue in calculating the total amount, thus helping in shaping the good image of the supermarket.

Last but not least, the owner could guide and nurture the newcoming staff to handle the inventory list easily through this supermarket inventory management system. The sort() feature could help in sorting the ID, price, and quantity according to numbers and sorting the name of the inventory according to alphabets. In this case, the newcomers would get used to the operation of the supermarket, familiar with the inventory available and prices for each product in the supermarket in a low to high order easily in a shorter time.

In a nutshell, the objective of developing a supermarket inventory management system is to create a robust, automated, and data-driven solution that optimizes inventory management, reduces costs, improves customer satisfaction, and enhances overall supermarket operations.

By achieving these objectives, the system enables supermarkets to achieve better profitability and stay competitive in the retail market.

CHAPTER II: LITERATURE REVIEW

2.1 OVERVIEW OF DATA STRUCTURE AND ALGORITHM

There are a few algorithms used in this project, which are separated into searching and sorting method. For the searching algorithm, linear search and binary search are involved, while for sorting algorithm, bubble sort and merge sort are included in this study. Despite that, there are two main data structure applied in designing and constructing our program, which are ArrayList and LinkedList data structure.

2.1.1 LINEAR SEARCH

First of all, let’s talk about the linear search method. Linear search is also known as sequential search algorithm, which the searching will start from either the head or the tail of an array containing a list of multiple elements, searching through the list one by one, comparing the target element and the elements in the list slowly until the desired elements is found, then the address of the matching target element is returned, else the searching process will continue until the end of the list. However, there are three different complexity cases when performing searching method, which are the best, worst and average case. For instance, the best-case complexity in linear search is that the desired element is found in the first position in the list, which means it does not require further searching and comparing process and ends with a single successful comparison, hence the best case of linear search algorithm presents 0(1) operations. Worst case scenario is completely reversed the concept of best-case scenario, as the desired element is placed at the last position of the list, thus it performs 0(N) operation. Lastly, the average case for linear search is when the desired element is in the middle of the array, therefore, it is performing 0(N) operations as well.

2.1.2 BINARY SEARCH

The another searching method used in this project is binary search. Binary search usually works in a sorted array and search the desired element by dividing the search interval into half repeatedly until the target element is found. The target element is compared with the mid element at the starting of the searching process, if the value of target element is lesser than mid element, then the elements on the left side of the mid elements is further proceed with the binary search again, else consider the right side of elements. This step is repeated continuously until the desired element is found, then it is finally terminated. The average time complexity of binary search algorithm is 0(log N). The best-case time complexity happened when the central index is directly matching with the desired element on the first try of searching, making the algorithm presenting 0(1) operations. Nonetheless, the worst case is that the desired element is placed at the first or last position in the list, as the list would require multiple searching and halving process to match with it, thus it performs 0(log N) operation.

2.1.3 BUBBLE SORT

There are 3 main sorting methods that we usually apply generally, including bubble sort, selection sort and insertion sort. However, in this project, bubble sort is used as the sorting algorithm to sort the items obtained from the created database accordingly. The way of how bubble sort works on sorting unsorted items are relatively easier and simpler comparing to other sorting methods, as it simply swaps the adjacent elements if they are placed in the wrong order in an array repeatedly, until the elements are arranged in an ascending or descending order accordingly. In bubble sort, the time-complexity is 0(N^2). The best-case complexity occurs when the elements or numbers is already sorted initially before performing the bubble sort, thus it requires no swap at all, while the number of comparisons is N-1, hence the best-case time complexity is 0(N). However, when the elements are arranged in a descending order in the array, it is known as the worst case as it requires the maximum number of comparisons and swapping process to sort the elements, hence the worst-case time complexity is 0(N^2).

2.1.4 MERGE SORT

Despite using bubble sort, advanced sort algorithm such as merge sort is utilised in constructing the program of our application too. Merge sort is a stable and reliable sorting algorithm which mainly works on the divide-and-conquer strategy. For example, it divides the array in half, this step is done recursively until all the subarrays have been divided into single elements, then the elements are compared one by one before placing them properly in position and merge all of them back to form a complete sorted array. In merge sort algorithm , the time complexity for average, best and worst case are all the same, which is 0(N log(N)), this is because, the algorithm will naturally apply the divide-and-conquer strategy when facing any kinds of array, no matter it is unsorted, sorted in ascending or descending order, the algorithm will still carry out the dividing and merging steps eventually.

2.1.5 ARRAY LIST

In java, there is a collection framework mainly containing List, Queue, and Set. In each interface, there are various classes such as ArrayList, LinkedList, Priority Queue, HashSet, and others, which every class has different functions and features. While in this project, ArrayList and LinkedList are used as the data structure to handle the data in database. First, ArrayList is a dynamic array-based data structure, unlike typical array, the size of ArrayList is not fixed, it provides an automatic resizable array implementation to increase or decrease its capacity dynamically when elements are added or removed from the list. The data stored in ArrayList can be accessed randomly and it can store any types of data including primitive and objects. In ArrayList, there are several common methods used, which are add(), remove(), get(), indexOf(), size(), isEmpty(), and others. Besides, the time complexity in ArrayList is different depends on the methods used, when add() is operating, the time-complexity is 0(1), while it is 0(N) when performing methods such as, get() and remove().

2.1.6 LINKED LIST

Despite ArrayList, LinkedList is another alternative data structure used to handle data in this project. LinkedList is one of the classes under List interface. LinkedList is a linear data structure which separate the elements into a data part and an address part in a single node. The elements are then linked together using pointer and address. LinkedList is a dynamic array-based data structure as well, thus the elements can be added into or removed from the list dynamically, as the size of the list will be adjusted automatically. In addition, LinkedList can be used to reduce memory wastage as the memory allocation is not contiguous like arrays, each node only stores a value and the reference to the next node. There are several basic methods found in LinkedList too, including add(), get(), remove(), clear(), isEmpty(), indexOf(), and others. The methods obtained from LinkedList are somehow similar to ArrayList as both of these data structure are under the same interface, List. Moreover, the time complexity is 0(1) when performing add(), and it is 0(N) when performing methods such as get() and remove() too.

2.2 JUSTIFICATION ON THE DATA STRUCTURE AND ALGORITHM

2.2.1 LINEAR SEARCH AND BINARY SEARCH

There are a few differences between the features of linear and binary search. In the aspect of data requirements, linear search could perform normally in both sorted and unsorted lists of elements, while binary search could only perform when the input data is sorted and cannot be implemented on unsorted data as the repeated targeting of the mid elements of one half depends on the sorted order of data structure. Besides, linear search functions in a slower manner but it is easier to implement and very straightforward in searching the target element, whilst binary search function is comparatively faster but it is more complex to implement too. In linear search, both single and multidimensional arrays can be used, but only single dimensional array can be used while implementing binary search. The efficiency of each algorithm depends on the characteristics of input data we are dealing with, for instance, if it is a sorted and huge dataset, then binary search is implemented as it significantly reduces the search space with each comparison, leading to faster search times. However, if the input data is relatively smaller and unsorted, then linear search is implemented as only linear search can be used to handle unsorted elements and it is more suitable for small size data too, where in this case simplicity is prioritised over efficiency.

2.2.2 BUBBLE SORT AND MERGE SORT

Bubble sort is a comparison-based sorting algorithm, while merge sort is a divide-and-conquer strategy-based algorithm, thus showing that both of this sorting algorithm implements in a completely different way. Generally, merge sort is considered as a stable and efficient sorting algorithm as it provides a consistent time complexity of 0(N log N) while handling and sorting a huge amount of dataset. However, bubble sort has minimal memory overhead as it does not require extra memory for auxiliary data structure, making it the preferred algorithm to handle smaller dataset, despite that bubble sort is inefficient for large dataset because of the time complexity of $0(N^2)$. In addition, bubble sort is an in-place sorting algorithm, which means it could amend the input elements without the needs of extra memory, but merge sort is an out-place sorting algorithm, it merges the sorted subarrays with additional memory. Practically, merge sort could be performing more efficiently than bubble sort while dealing with real life sorting application, as usually we would need to deal with a large amount of data and in this case, merge sort would be the better options selected.

2.2.3 ARRAY LIST AND LINKED LIST

ArrayList and LinkedList can be considered as the most frequently used classes under list interface. (Anant Mishra, 2019). ArrayList performs based on the array data structure, while LinkedList uses the DoublyLinkedList data structure, hence the access time for random data access in ArrayList is much faster than LinkedList. Furthermore, the elements stored in the ArrayList consecutively making it easy to be accessed directly by relying on their index, while in LinkedList, it requires to access multiple memories to traverse the link nodes from the beginning because the elements are scattered throughout memory in LinkedList. In addition, because of the scattered nature of the nodes in memory, it leads to a slower process in iterating over elements with iterator in LinkedList than in ArrayList. Besides, LinkedList only requires necessary memory for the data and 2 references in each node, while ArrayList requires extra spaces for the resizing of array, thus LinkedList has a relatively lower memory overhead. Last but not least, in terms of the efficiency in both data structures, LinkedList would perform efficiently if there are numerous insertion and deletion operation to be conducted and the order of elements are changing oftenly. However, if the order of elements is stable, fast random access is required and the operations of insertion and deletion is comparatively limited, then ArrayList is selected in this case.

CHAPTER III: METHODOLOGY

3.1 SYSTEM UML DIAGRAM

ARRAYLIST + BINARY SEARCH + BUBBLE SORT

A system UML diagram offers a high-level overview of the architecture, parts, and interactions of the system. It aids in the behaviour and structural visualization of the system. The system UML diagram for the provided code will show the key elements, their connections, and the data and control flow during program execution.

1. Components:
   - Main Class (Binary Names): This represents the system's central element, which coordinates interactions between other elements and contains the main method.
   - File Operations: This represents the part of the system in charge of reading and writing data to files.
   - Sorting: The component that sorts the data (Bubble Sort) is represented here.

- Searching: This represents the part of the system that searches for binary patterns in the sorted data.
- User Interface: This represents the part in charge of communicating with users and showing information.

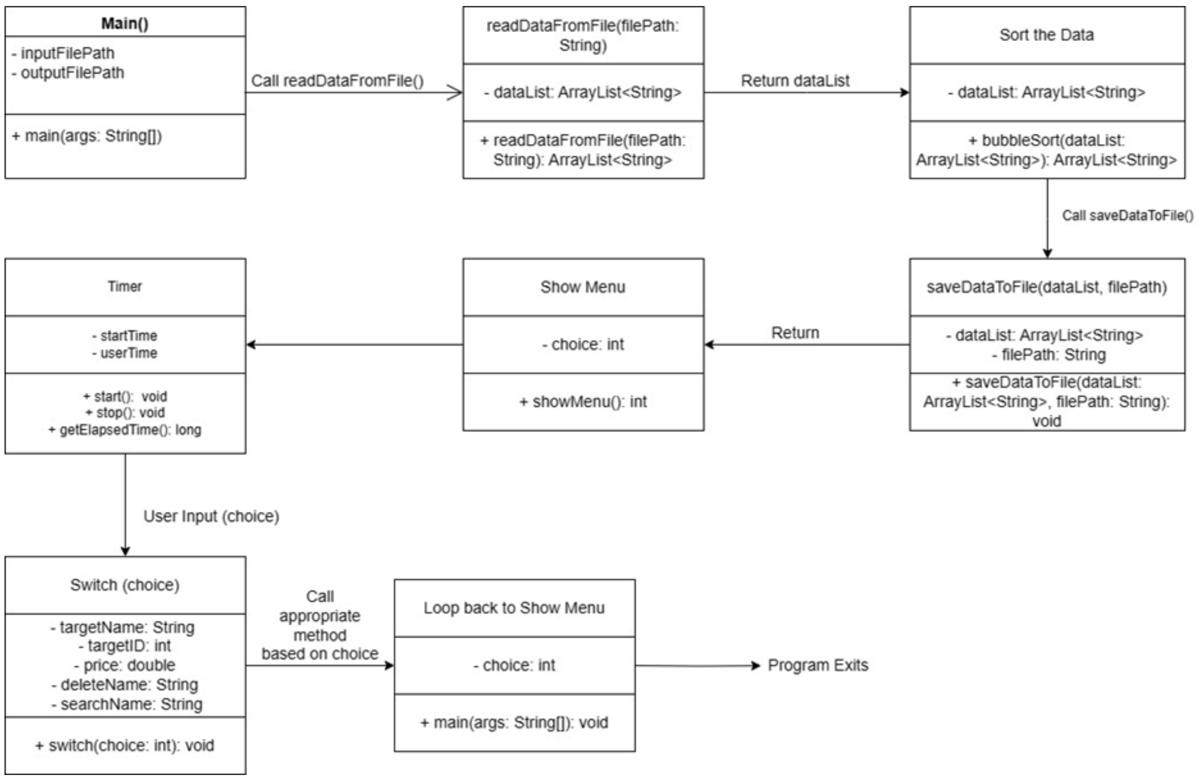


Figure 1 System bubble sort UML diagram

2. System UML Diagram Considerations:
   - With a focus on high-level interactions, the system UML diagram offers an abstract illustration of the system's parts and their connections.
   - If the code is a part of a bigger system, the diagram can be enlarged to include more elements or interactions.

LINKED LIST + LINEAR SEARCH + MERGE SORT

1- Main Class (Name):
   - The primary program entrance point is represented by this component.
   - It manages the application's general flow, including reading data from the database, conducting searches, and sorting, and interacting with users.

- To perform its functions, the main class interacts with other components, such as the File Operations component for reading and writing files and the Data Management component for manipulating data.

2- File Operations:

- This part is in charge for both reading and writing data to the database file.
- During program initialization, it offers ways to read product details (ID, name, and price) into a data structure (LinkedList of Product objects) from the database file.
- After sorting operations, it also stores the sorted data back to the database file.
- Data durability between different program runs is made possible with the aid of the File Operations component.

3- Data Management:

- This component manages the operations of the product data such as data processing, sorting, and searching.
- It includes techniques for using the Merge Sort algorithm to arrange the product data according to product IDs.
- Additionally, it offers ways to conduct a linear search for products using their names.
- Data management makes sure that the organization and accessibility of product data is effective.

4- User Interface:

- The user interface element communicates with the user and presents data.
- It solicits information from the user and asks them to select an action from a list of possibilities.
- It interacts with the Main Class to carry out the requested action, such as adding a product, deleting a product, looking for a product, or quitting the program, according to the user's choice.
- The user interface shows the results or the relevant notifications to the user after performing the selected action.

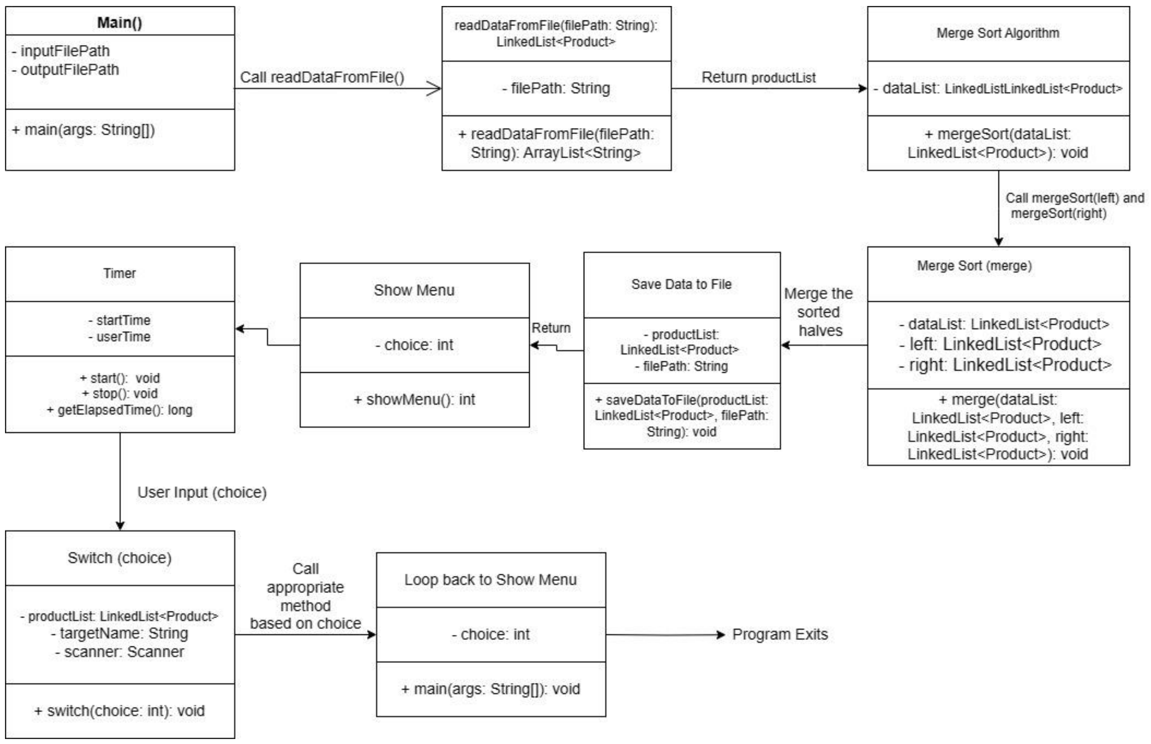

Main()
- inputFilePath
- outputFilePath
+ main(args: String[])
Call readDataFromFile()
readDataFromFile(filePath: String): LinkedList<Product>
- filePath: String
+ readDataFromFile(filePath: String): ArrayList<String>
Return productList
Merge Sort Algorithm
- dataList: LinkedListLinkedList<Product>
+ mergeSort(dataList: LinkedList<Product>): void
Call mergeSort(left) and mergeSort(right)
Merge Sort (merge)
- dataList: LinkedList<Product>
- left: LinkedList<Product>
- right: LinkedList<Product>
+ merge(dataList: LinkedList<Product>, left: LinkedList<Product>, right: LinkedList<Product>): void
Merge the sorted halves
Save Data to File
- productList: LinkedList<Product>
- filePath: String
+ saveDataToFile(productList: LinkedList<Product>, filePath: String): void
Return
Show Menu
- choice: int
+ showMenu(): int
Timer
- startTime
- userTime
+ start(): void
+ stop(): void
+ getElapsedTime(): long
User Input (choice)
Switch (choice)
- productList: LinkedList<Product>
- targetName: String
- scanner: Scanner
+ switch(choice: int): void
Call appropriate method based on choice
Loop back to Show Menu
- choice: int
+ main(args: String[]): void
Program Exits

Figure 2 System Merge sort UML Diagram

The relationships between components are depicted as follows:

- Because the File Operations component is dependent upon the Main Class, the Main Class depends on File Operations in order to read and write data to the database file. When necessary, it invokes the relevant methods from the File Operations component.
- The fact that the Main Class is also dependent on the Data Management component shows how the Main Class uses Data Management to manipulate, sort, and search data. To complete these duties, it invokes the pertinent methods from the Data Management component.
- The Main Class is dependent on the User Interface component, meaning that the User Interface uses the Main Class to execute commands and receive data. Through method calls, it communicates with the Main Class and executes user-selected actions.
- The User Interface component and the Data Management component are connected, proving that the Data Management component offers services to the User Interface. It makes it possible for the User Interface to carry out data-manipulation operations like sorting and searching.

3.2 USE CASE DIAGRAM

In a software application, a use case diagram depicts the functional needs and interactions between actors and the system. The main use case for the provided code entails controlling a product database through various system interactions. The "User" and the "Product Database System" are the main players in this scenario.

1. Actors:
   - User: embodies the user of the system's product database. The user interacts with the system to carry out tasks including adding, removing, and searching for certain products.
2. Use Cases:
   - Add Product: By entering the product's ID, name, and price, the user can add a new item to the database.
   - Product deletion: By entering the product's ID, the user can remove a product from the database.
   - Product search: A user can look for a product by entering its name.
   - View Sorted Data: The system uses the Merge Sort algorithm to sort the product data before showing it to the user.
3. Relationships:
   - <<include>>: Other use cases, such as "Add Product" and "Delete Product," also include the "View Sorted Data" use case. This means that the system will automatically sort the data before displaying it to the user whenever the user adds or deletes a product.
4. Use Case Diagram Notation: The use case diagram consists of actors represented as stick figures, use cases as ovals with names, and relationships between actors and use cases represented by arrows.

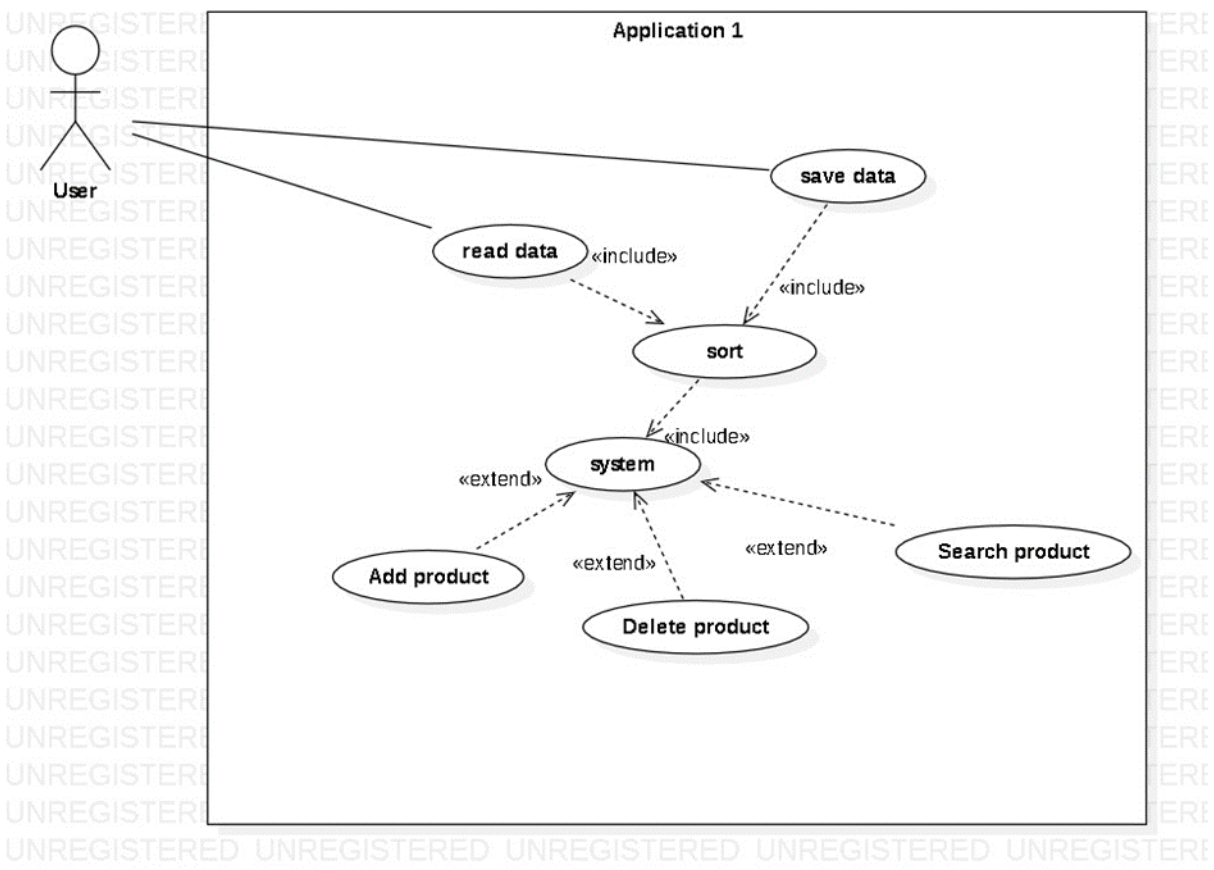


Figure 3 UseCase diagram

3.3 SYSTEM FLOWCHART

ARRAYLIST + BINARY SEARCH + BUBBLE SORT for Names and IDs:

1. Problem Statement: A Java program is included in the source code that carries out a number of operations on a list of product names, IDs, prices, and quantities. The program's primary features include reading data from an input file, sorting the data according to product names, adding new goods, removing current items, and doing binary searches for specific products. The program also monitors how much memory and runtime are used by the primary operations.

2. Input and Output Specification: The programme reads information from an input file that is formatted as "ID, Name, Price, Quantity" for each product, with each line denoting a different product. After sorting the data according to product names, the output is saved to a new file.

3. Approach:

The following methodology can be used to describe this Java programme:

i. Data Reading: An ArrayList of strings is used to hold the data that the programme reads from the input file. The format "ID, Name, Price, Quantity" is used to identify each element of the ArrayList as a product.

ii. Sorting: The Bubble Sort algorithm is used to arrange the data according to the names of the products. The ArrayList is sorted in ascending order using the bubbleSort() method.

iii. Adding a product: By providing the product name, ID, price, and quantity, users can add new products. If the product name doesn't already exist, the programme adds the new product to the ArrayList. The list is resorted after addition.

iv. Deleting a Product: By entering the product name, users can delete products. The programme looks to see if the product name is already present in the ArrayList and eliminates it if it is. The list is then revised and reorganised.

v. Searching a Product: Users can conduct a product search by entering the item's name. To locate the product name, the programme uses binary search on the sorted ArrayList. If a match is made, the product's information (ID, cost, and quantity) is displayed.

vi. Runtime and Memory Measurement: The add, delete, and search methods' execution times are tracked by the programme using the Timer class. Using the getMemoryUsedDuringMethod() method, it also calculates the amount of memory utilised for each of these methods.

4. File Paths: The main() method has the paths to the input and output files hardcoded. For the programme to operate properly, the user must provide the necessary file paths.

5. Complexity Analysis:

   i. Reading Data: O(n), where n is the number of lines in the input file, is the complexity for reading data.

   ii. Sorting: In the worst scenario, the complexity of the Bubble Sort algorithm is O(n2).

   iii. Adding and Deleting: Due to the linear search used to discover the product by name, the complexity for adding and removing a product from the ArrayList is O(n).

   iv. Searching: Binary search has an O(log n) complexity, where n is the number of items in the array list.

6. Potential Improvements:

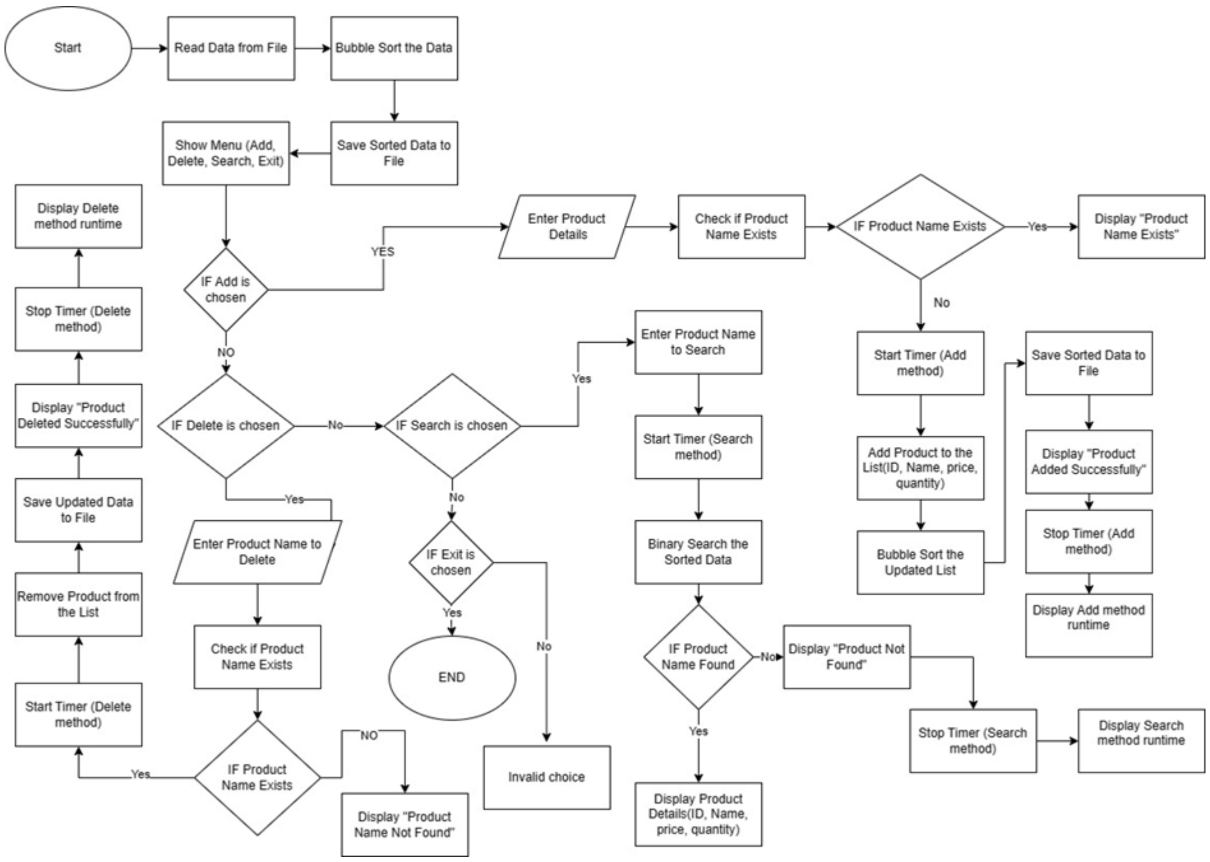
   i. Consider adopting more effective sorting algorithms like QuickSort or MergeSort, which have superior average and worst-case complexity, to boost sorting speed.

   ii. To manage incorrect user inputs and avoid unexpected behaviour, implement data validation.

   iii. Add error and exception handling to effectively manage unforeseen circumstances.

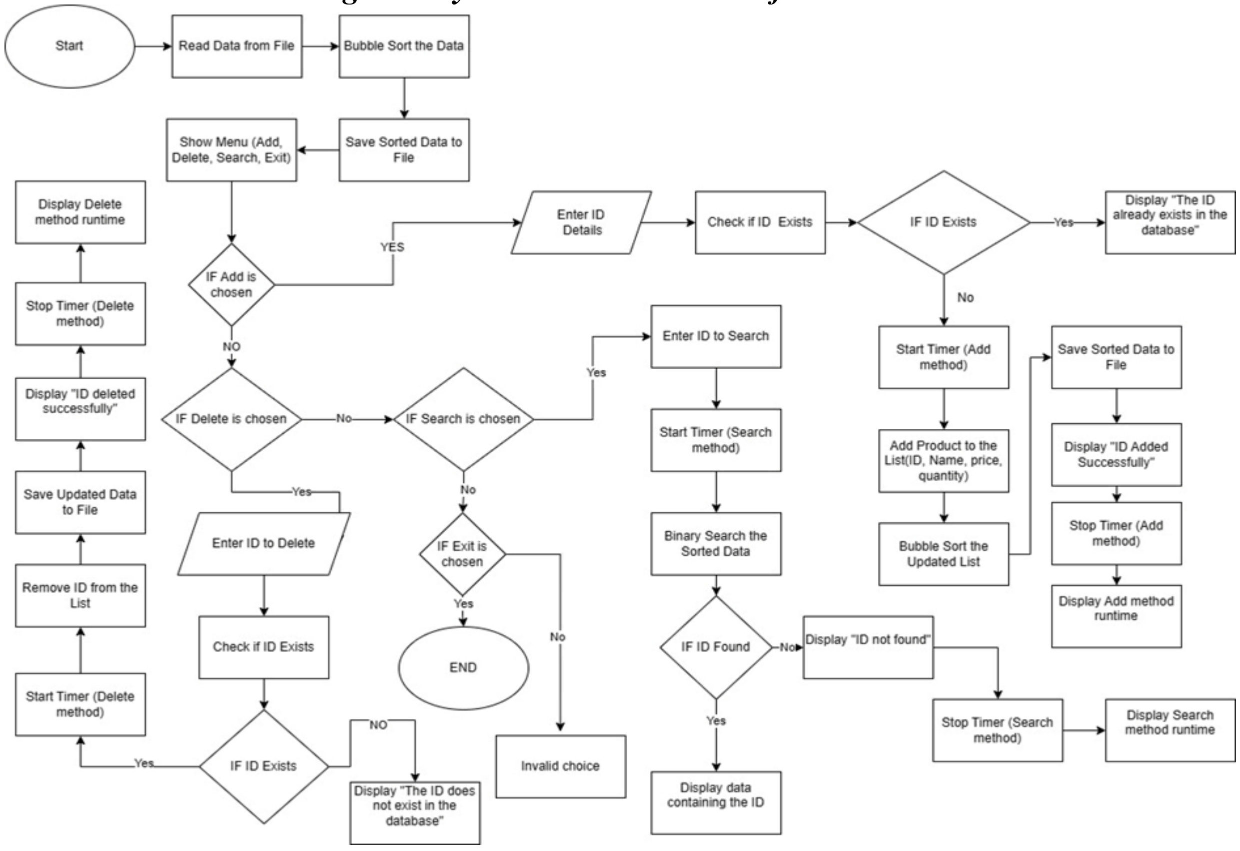
   iv. To improve the user experience, offer clear error messages and instructions.

7. Testing: Test the programme thoroughly using a variety of scenarios, such as adding, deleting, and searching for products. To evaluate the program's performance and memory utilisation, run it through a large number of input tests. Check to see if the sorted data is in the output file and that the add, remove, and search features perform as expected.



Start
Read Data from File
Bubble Sort the Data
Save Sorted Data to File
Show Menu (Add, Delete, Search, Exit)
IF Add is chosen
YES
Enter Product Details
Check if Product Name Exists
IF Product Name Exists
Yes
Display "Product Name Exists"
No
Start Timer (Add method)
Add Product to the List(ID, Name, price, quantity)
Bubble Sort the Updated List
Save Sorted Data to File
Display "Product Added Successfully"
Stop Timer (Add method)
Display Add method runtime
NO
IF Delete is chosen
No
IF Search is chosen
Yes
Enter Product Name to Search
Start Timer (Search method)
Binary Search the Sorted Data
IF Product Name Found
No
Display "Product Not Found"
Stop Timer (Search method)
Display Search method runtime
Yes
Display Product Details(ID, Name, price, quantity)
No
IF Exit is chosen
Yes
END
No
Invalid choice
Yes
Enter Product Name to Delete
Check if Product Name Exists
IF Product Name Exists
NO
Display "Product Name Not Found"
Yes
Start Timer (Delete method)
Remove Product from the List
Save Updated Data to File
Display "Product Deleted Successfully"
Stop Timer (Delete method)
Display Delete method runtime

Figure 4 System bubble sort name flowchart



Start
Read Data from File
Bubble Sort the Data
Save Sorted Data to File
Show Menu (Add, Delete, Search, Exit)
IF Add is chosen
YES
Enter ID Details
Check if ID Exists
IF ID Exists
Yes
Display "The ID already exists in the database"
No
Start Timer (Add method)
Add Product to the List(ID, Name, price, quantity)
Bubble Sort the Updated List
Save Sorted Data to File
Display "ID Added Successfully"
Stop Timer (Add method)
Display Add method runtime
NO
IF Delete is chosen
No
IF Search is chosen
Yes
Enter ID to Search
Start Timer (Search method)
Binary Search the Sorted Data
IF ID Found
No
Display "ID not found"
Stop Timer (Search method)
Display Search method runtime
Yes
Display data containing the ID
No
IF Exit is chosen
Yes
END
No
Invalid choice
Yes
Enter ID to Delete
Check if ID Exists
IF ID Exists
NO
Display "The ID does not exist in the database"
Yes
Start Timer (Delete method)
Remove ID from the List
Save Updated Data to File
Display "ID deleted successfully"
Stop Timer (Delete method)
Display Delete method runtime

Figure 5 System bubble sort IDs flowchart

LINKED LIST + LINEAR SEARCH + MERGE SORT for Names and IDs:

1. Problem Statement: A database of items with distinct IDs, names, and prices is maintained by the Java program that is provided. Users of the program can add a new product, remove an existing one by its ID, and search for a product by name, among other activities on the database. The data is sorted using Merge Sort in the program and is then saved in a file for persistence.

2. Input and Output: The program pulls information from a file with product details in the order "ID, Name, Price" for each product, where each line corresponds to a different product. After updating and sorting, the output is saved to the same file.

3. Approach:

   The methodology for this Java program can be described as follows:

   a) Reading Data: Data from the input file is read by the program, which then stores it in a LinkedList of Product objects. With fields like ID, name, and price, each Product object represents a single product.

   b) Sorting: Based on product IDs, the data is sorted using the merge sort method. The LinkedList of Product objects is sorted by the mergeSort() function.

   c) Adding a Product: By providing the product ID, name, and price, users can add new products. The program determines whether the ID is unique or whether it is already present in the database before adding the new product. The product is added, and then the LinkedList is sorted once more using Merge Sort.

   d) Deleting a Product: By inputting the product ID, users can delete a product. The program searches the LinkedList for the product with the supplied ID and, if it is found, eliminates it. Following that, the LinkedList is modified and saved to the database file.

   e) Searching a Product: Users can input a product's name to search for it. For the purpose of locating the item with the specified name, the program runs a linear search on the LinkedList. If a match is made, the product information (ID, name, and price) is displayed.

f) File Handling: To read and write data from and to the database file, the program employs file handling. It guarantees that the product IDs are consistently formatted with four digits.

g) Complexity Analysis:
   - Reading Data: O(n), where n is the number of lines in the file, is the complexity of reading data from it.
   - Sorting: Merge Sort has an O(n log n) complexity, where n is the number of items in the LinkedList.
   - Adding and Deleting: Due to the linear search used to discover the product by ID, the complexity for adding and deleting a product is O(n).
   - Searching: When there are n items in the LinkedList, the complexity of a linear search is O(n).

4. Potential Improvements:
   - Consider storing products in more effective data structures like a HashMap, where product names serve as the keys and the actual products as the values, to improve search performance.
   - To manage incorrect user inputs and avoid unexpected behavior, implement data validation.
   - Add error and exception handling to effectively manage unforeseen circumstances.
   - To improve the user experience, offer clear error messages and instructions.
   - Think about allowing consumers to sort the data based on additional criteria besides ID, such as name or price.
5. Testing: Test the program thoroughly using a variety of scenarios, such as adding, deleting, and searching for products. To evaluate the program's performance and memory utilization, run it through a number of data-input tests. Make sure the add, remove, and search capabilities operate as intended and that the database file contains the sorted data.

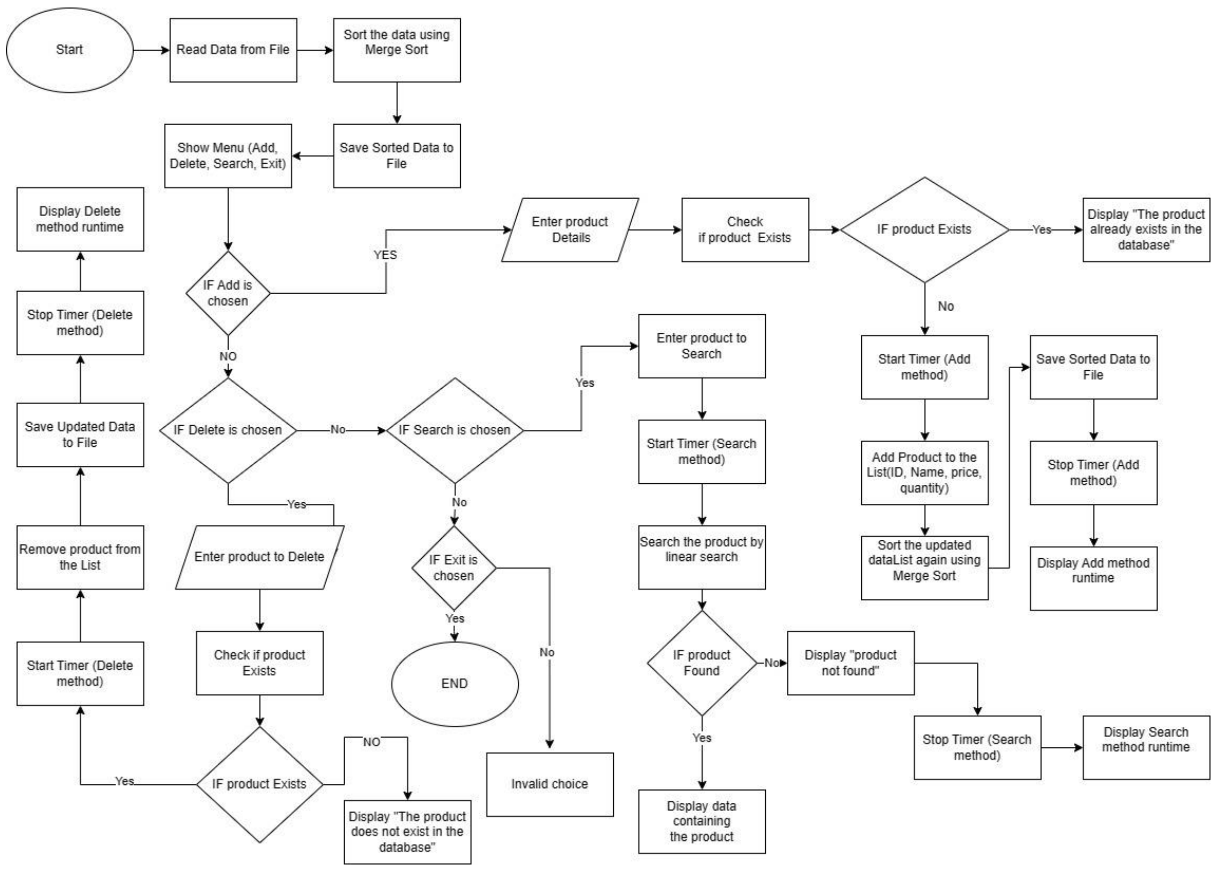

Start
Read Data from File
Sort the data using Merge Sort
Save Sorted Data to File
Show Menu (Add, Delete, Search, Exit)
IF Add is chosen
YES
NO
Enter product Details
Check if product Exists
IF product Exists
Yes
Display "The product already exists in the database"
No
Start Timer (Add method)
Add Product to the List(ID, Name, price, quantity)
Sort the updated dataList again using Merge Sort
Save Sorted Data to File
Stop Timer (Add method)
Display Add method runtime
IF Delete is chosen
IF Search is chosen
Enter product to Search
Start Timer (Search method)
Search the product by linear search
IF product Found
Display "product not found"
Stop Timer (Search method)
Display Search method runtime
Display data containing the product
IF Exit is chosen
END
Invalid choice
Enter product to Delete
Check if product Exists
IF product Exists
Display "The product does not exist in the database"
Start Timer (Delete method)
Remove product from the List
Save Updated Data to File
Stop Timer (Delete method)
Display Delete method runtime

Figure 6 System Merge sort names flowchat

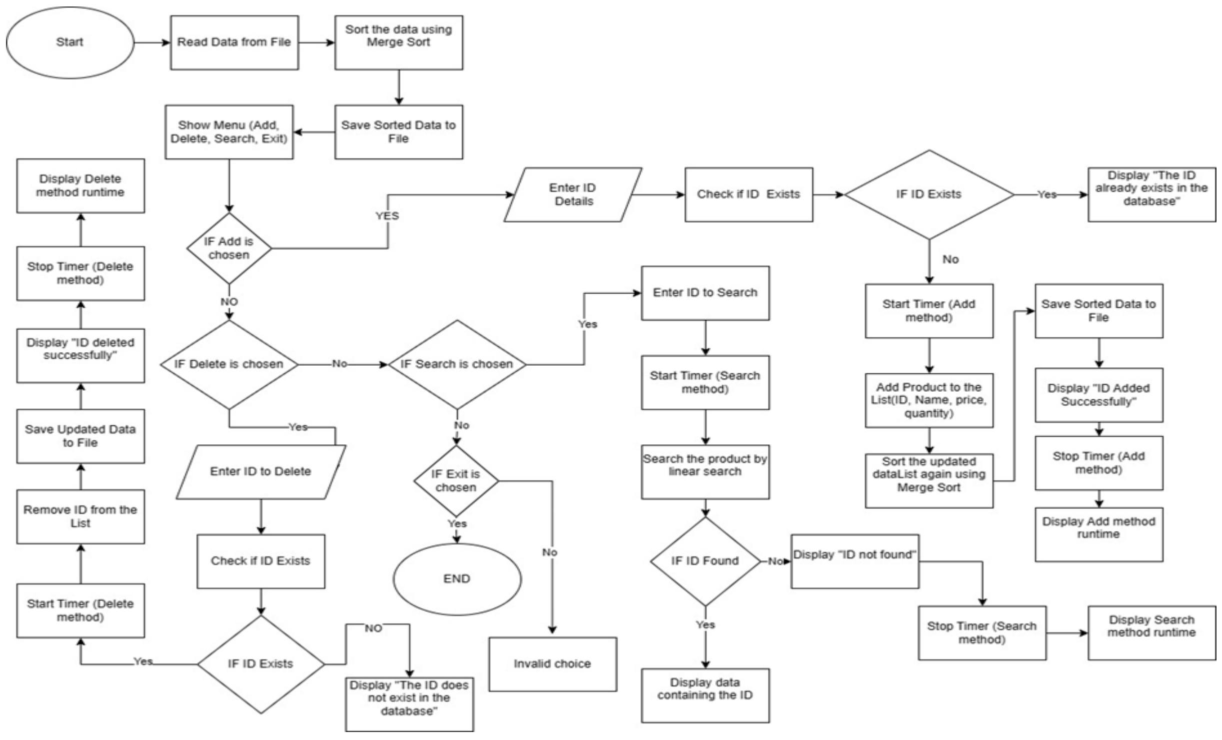

Start
Read Data from File
Sort the data using Merge Sort
Save Sorted Data to File
Show Menu (Add, Delete, Search, Exit)
IF Add is chosen
YES
NO
Enter ID Details
Check if ID Exists
IF ID Exists
Yes
Display "The ID already exists in the database"
No
Start Timer (Add method)
Add Product to the List(ID, Name, price, quantity)
Sort the updated dataList again using Merge Sort
Save Sorted Data to File
Display "ID Added Successfully"
Stop Timer (Add method)
Display Add method runtime
IF Delete is chosen
IF Search is chosen
Enter ID to Search
Start Timer (Search method)
Search the product by linear search
IF ID Found
Display "ID not found"
Stop Timer (Search method)
Display Search method runtime
Display data containing the ID
IF Exit is chosen
END
Invalid choice
Enter ID to Delete
Check if ID Exists
IF ID Exists
Display "The ID does not exist in the database"
Start Timer (Delete method)
Remove ID from the List
Save Updated Data to File
Display "ID deleted successfully"
Stop Timer (Delete method)
Display Delete method runtime

Figure 7 System Merge sort IDs flowchart

CHAPTER IV: RESULT AND DISCUSSION

4.1 APPLICATION MODULE CODE

4.1.1 APPLICATION 1: ARRAYLIST + BINARY SEARCH + BUBBLE SORT

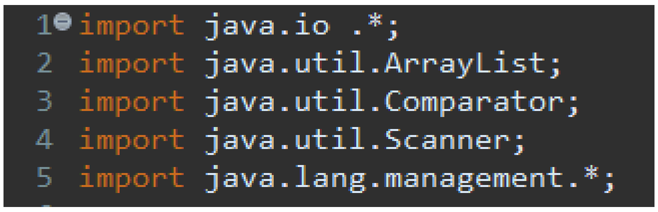
By using the same Database, we will work on different data from it using two different codes:

I. One code: sort, add, delete and search using ID numbers product, and save all modifications of this database in a new file.

II. And the other code: sort, add, delete and search using Names of product, and save all modifications of this database in a new file.

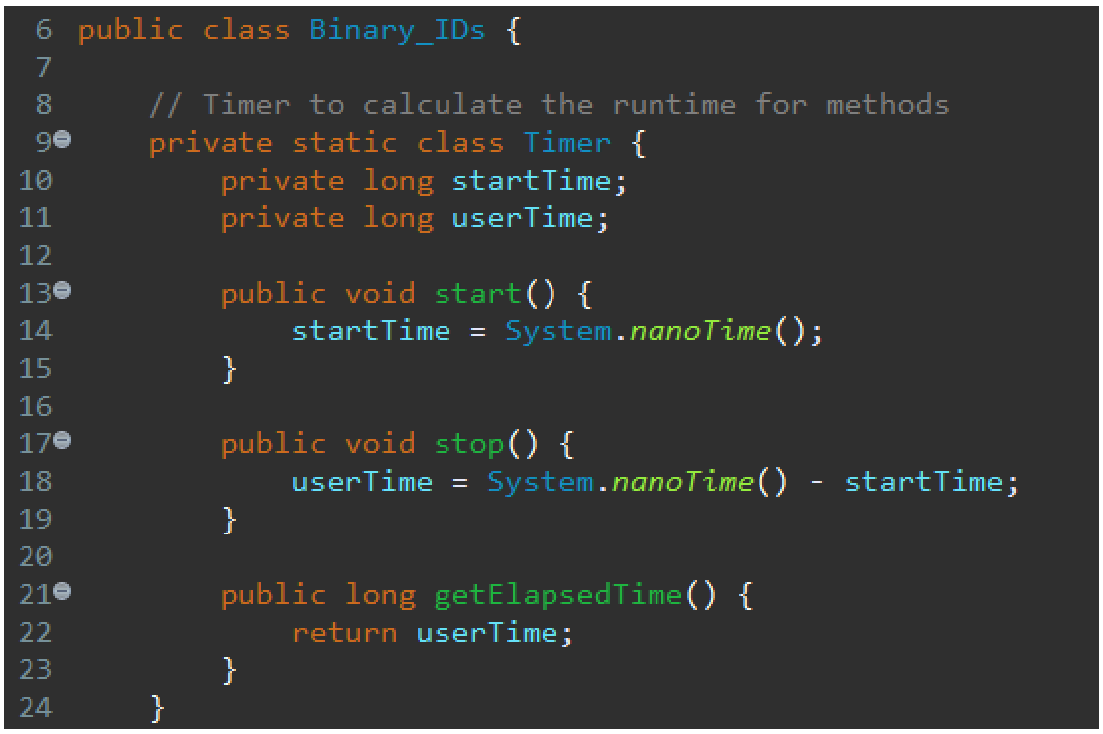
I. **Product ID Application 1: Arraylist + binary search + Bubble sort**:

We will start by explaining the code then show the program running. We will divide the code into different parts so it will be easy to explain and to understand.



```java
import java.io .*;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.Scanner;
import java.lang.management.*;
```

We start by importing all the packages, sub-packages, classes, interfaces we need for the code.



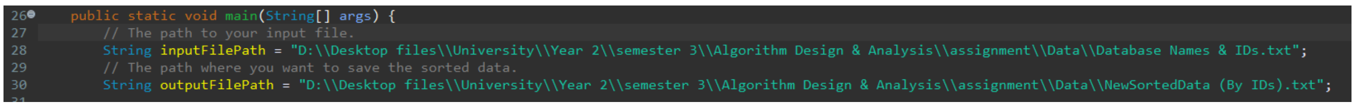
```java
public class Binary_IDs {

    // Timer to calculate the runtime for methods
    private static class Timer {
        private long startTime;
        private long userTime;

        public void start() {
            startTime = System.nanoTime();
        }

        public void stop() {
            userTime = System.nanoTime() - startTime;
        }

        public long getElapsedTime() {
            return userTime;
        }
    }
```

So, the name of the class is “*Binary_IDs*”

We started by making a timer class that calculates the run time of the program without considering the time taken by the user to enter data. Along the explanation of the code, we will find that we have used this timer method to calculate the runtime of each task the user chooses to execute (search, add, delete).

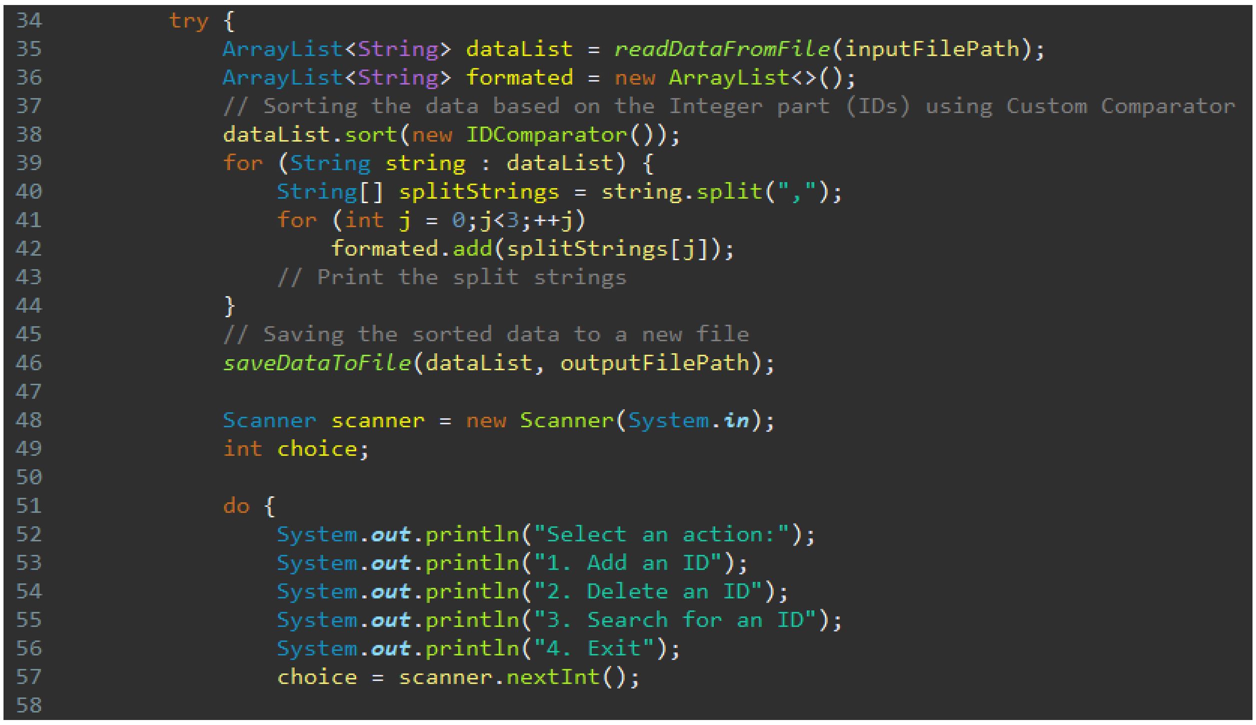
Inside this timer class, we have 3 methods:

1. *start()* : to start the time
2. *stop()* : to calculate the final time after it removes the time the user takes to put the input
3. *getElapsedTime()* : it returns the runtime



```java
    public static void main(String[] args) {
        // The path to your input file.
        String inputFilePath = "D:\\Desktop files\\University\\Year 2\\semester 3\\Algorithm Design & Analysis\\assignment\\Data\\Database Names & IDs.txt";
        // The path where you want to save the sorted data.
        String outputFilePath = "D:\\Desktop files\\University\\Year 2\\semester 3\\Algorithm Design & Analysis\\assignment\\Data\\NewSortedData (By IDs).txt";
```

Here, we started writing in the main function (the one that program runs). We indicated the path (location in your PC) for database (input file) and the path for the new updated or edited database (output file).

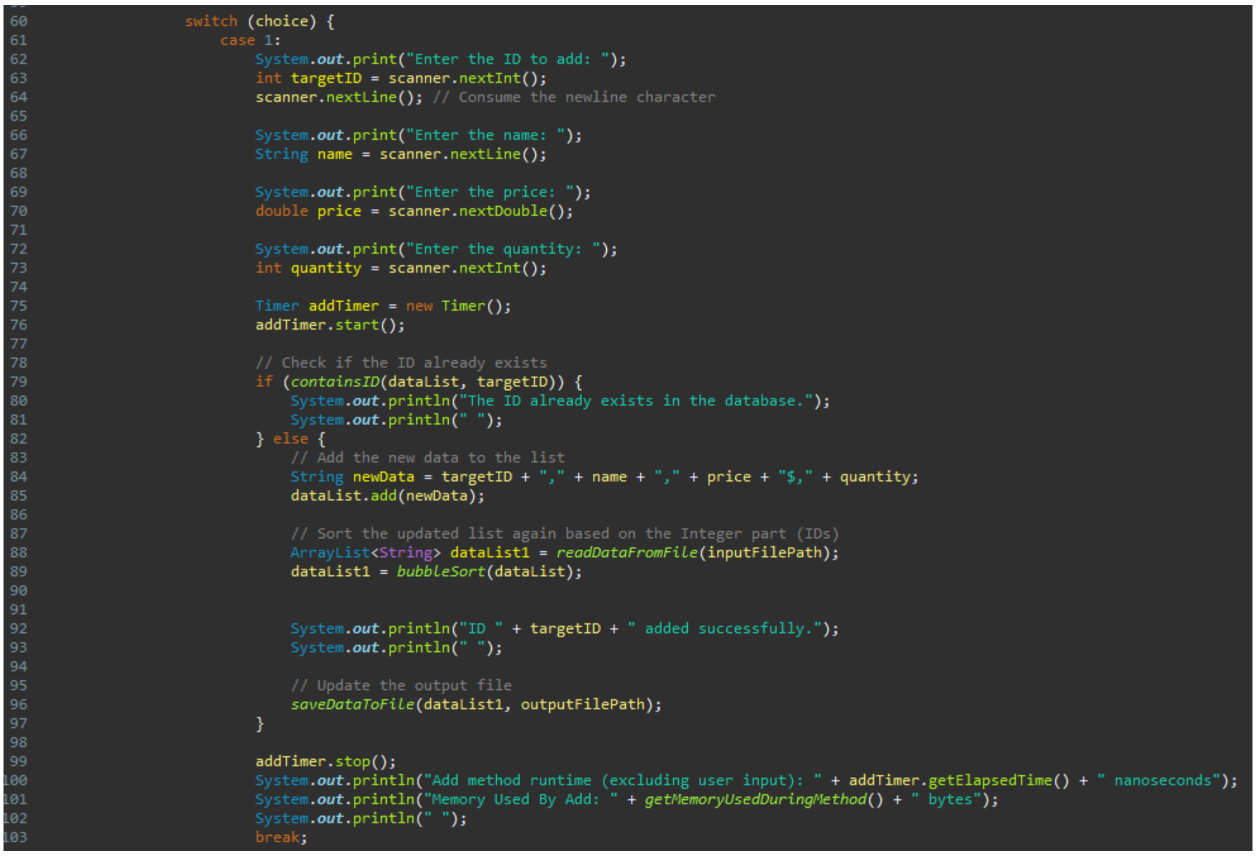


```java
        try {
            ArrayList<String> dataList = readDataFromFile(inputFilePath);
            ArrayList<String> formated = new ArrayList<>();
            // Sorting the data based on the Integer part (IDs) using Custom Comparator
            dataList.sort(new IDComparator());
            for (String string : dataList) {
                String[] splitStrings = string.split(",");
                for (int j = 0;j<3;++j)
                    formated.add(splitStrings[j]);
                // Print the split strings
            }
            // Saving the sorted data to a new file
            saveDataToFile(dataList, outputFilePath);

            Scanner scanner = new Scanner(System.in);
            int choice;

            do {
                System.out.println("Select an action:");
                System.out.println("1. Add an ID");
                System.out.println("2. Delete an ID");
                System.out.println("3. Search for an ID");
                System.out.println("4. Exit");
                choice = scanner.nextInt();

```

In this part of code, the code reads data from a file specified by the variable *inputFilePath* and stores it in an *ArrayList* named *dataList*. Then the data in *dataList* is firstly sorted based on the integer part (IDs) using a custom comparator called *IDComparator*. The *IDComparator* class implements the Comparator interface to define a custom sorting order for the data. After sorting, the code extracts the first three elements from each string in the *dataList*, separated by commas *(,)*. These elements are then added to a new *ArrayList* named *formated*. Next, the sorted data in *dataList* is saved to a new file specified by the variable *outputFilePath* using the *saveDataToFile* method. After the data is sorted and saved, the code creates a Scanner object to interact with the user. It presents a menu to the user with four options:

- Option 1: Add an ID
- Option 2: Delete an ID
- Option 3: Search for an ID
- Option 4: Exit

The code enters a do-while loop to continuously prompt the user for input until they choose to exit (select option 4). Depending on the user's choice (1, 2, or 3), additional actions could be implemented to add, delete, or search for an ID in the data.



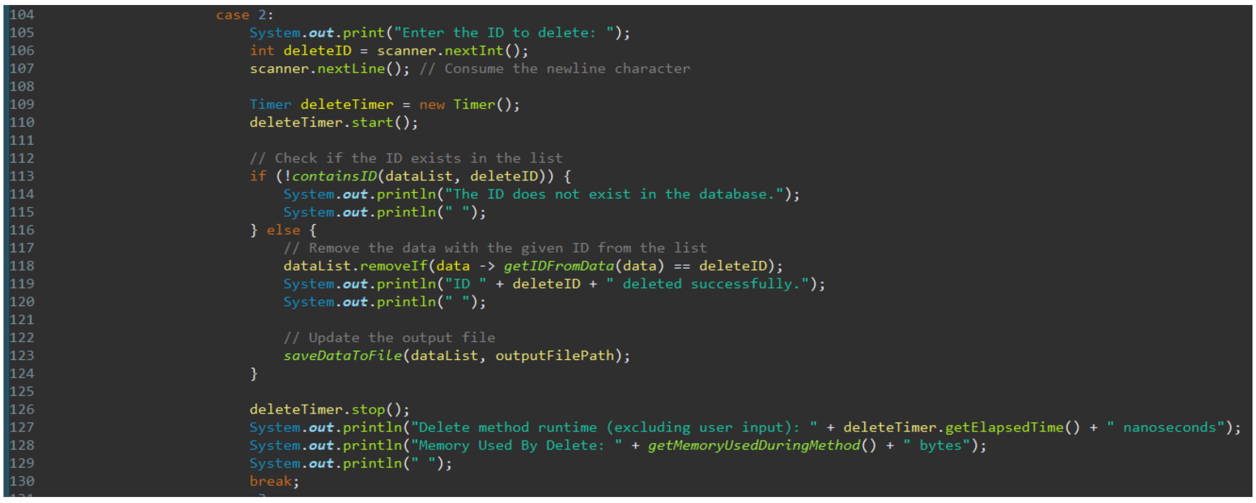
```java
            switch (choice) {
                case 1:
                    System.out.print("Enter the ID to add: ");
                    int targetID = scanner.nextInt();
                    scanner.nextLine(); // Consume the newline character

                    System.out.print("Enter the name: ");
                    String name = scanner.nextLine();

                    System.out.print("Enter the price: ");
                    double price = scanner.nextDouble();

                    System.out.print("Enter the quantity: ");
                    int quantity = scanner.nextInt();

                    Timer addTimer = new Timer();
                    addTimer.start();

                    // Check if the ID already exists
                    if (containsID(dataList, targetID)) {
                        System.out.println("The ID already exists in the database.");
                        System.out.println(" ");
                    } else {
                        // Add the new data to the list
                        String newData = targetID + "," + name + "," + price + "$," + quantity;
                        dataList.add(newData);

                        // Sort the updated list again based on the Integer part (IDs)
                        ArrayList<String> dataList1 = readDataFromFile(inputFilePath);
                        dataList1 = bubbleSort(dataList);


                        System.out.println("ID " + targetID + " added successfully.");
                        System.out.println(" ");

                        // Update the output file
                        saveDataToFile(dataList1, outputFilePath);
                    }

                    addTimer.stop();
                    System.out.println("Add method runtime (excluding user input): " + addTimer.getElapsedTime() + " nanoseconds");
                    System.out.println("Memory Used By Add: " + getMemoryUsedDuringMethod() + " bytes");
                    System.out.println(" ");
                    break;
```

Inside this case 1 block, the program prompts the user to enter the following information:

- An integer value for the ID to add.
- A string for the name associated with the ID.
- A double value for the price associated with the ID.

- An integer value for the quantity of the product.

The code checks if the entered ID already exists in the *dataList* (which contains the sorted data read from the file earlier). This happens by calling the *containsID* method (explained later). The purpose is to prevent adding duplicate IDs to the list. If the entered ID does not exist in the *dataList*, the code proceeds to add the new data to the list. It creates a new string *newData* that combines the entered ID, name, and price in the format: "ID,name,price$,quantity". For example, "990,Apple,25.5$,13". After adding the new data, the code reads the data from the input file again and sorts the updated list *dataList1* based on the integer part (IDs). It does this using the *bubbleSort* method (explained later). The purpose is to maintain the sorted order of the data. The updated and sorted *dataList1* is then saved to the output file specified by *outputFilePath* using the *saveDataToFile* method. This ensures that the changes are reflected in the file. The code measures the runtime of the "Add" method (excluding user input) using a Timer class. It prints the elapsed time in nanoseconds. And also print the memory space used by the function using *getMemoryUsedDuringMethod*(). This code handles the process of adding a new entry (ID, name, and price) to the data list, checks for duplicates, sorts the list, saves the updated list to the output file, and provides runtime and memory used too.



```java
case 2:
    System.out.print("Enter the ID to delete: ");
    int deleteID = scanner.nextInt();
    scanner.nextLine(); // Consume the newline character

    Timer deleteTimer = new Timer();
    deleteTimer.start();

    // Check if the ID exists in the list
    if (!containsID(dataList, deleteID)) {
        System.out.println("The ID does not exist in the database.");
        System.out.println(" ");
    } else {
        // Remove the data with the given ID from the list
        dataList.removeIf(data -> getIDFromData(data) == deleteID);
        System.out.println("ID " + deleteID + " deleted successfully.");
        System.out.println(" ");

        // Update the output file
        saveDataToFile(dataList, outputFilePath);
    }

    deleteTimer.stop();
    System.out.println("Delete method runtime (excluding user input): " + deleteTimer.getElapsedTime() + " nanoseconds");
    System.out.println("Memory Used By Delete: " + getMemoryUsedDuringMethod() + " bytes");
    System.out.println(" ");
    break;
```

Inside this case 2 block, this code is part of the switch statement that handles the user's choice for "Delete" operation in the menu.

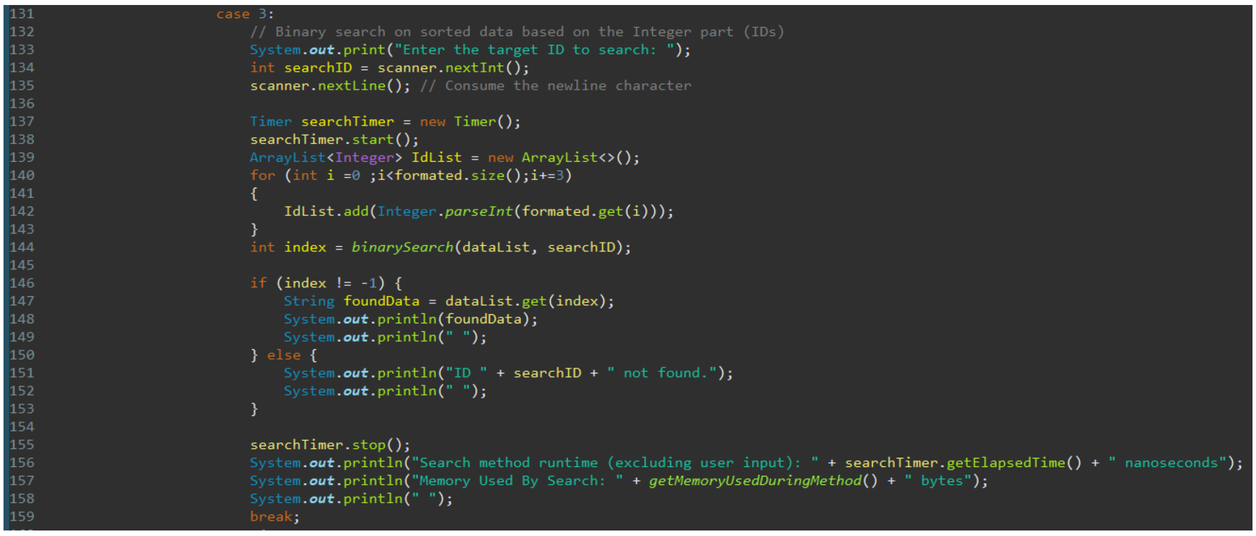
It prompts the user to enter the ID they want to delete from the database. It reads the user's input for the ID using *scanner.nextInt()*. After reading the integer, it consumes the newline character left in the input buffer by calling *scanner.nextLine()*. This is done to prepare for further input processing.A Timer object named *deleteTimer* is created and started to measure the time taken for the delete operation (runtime excluding user input).

It checks if the ID exists in the *dataList* by calling the *containsID* method. If the ID does not exist, it prints a message stating that the ID does not exist in the database. If the ID exists in the *dataList*, it removes all data entries (strings) that have the given ID from the list. It does this using the *removeIf* method, which uses a lambda expression to test each data entry's ID and removes it if the ID matches the *deleteID*.

If data entries with the given ID are found and removed from the *dataList*, it prints a success message indicating that the ID was deleted successfully. After deletion, it updates the output file by calling the *saveDataToFile* method, which writes the modified *dataList* back to the output file.

The *deleteTimer* is stopped, and the elapsed time for the delete operation (runtime excluding user input) is printed. And also print the memory space used by the function using *getMemoryUsedDuringMethod*().

Finally case 2 block ends with a break statement to exit the block. Overall, this code handles the process of deleting data entries from the *dataList* based on a given ID provided by the user. It ensures that the ID exists in the list before deleting it and then updates both the in-memory *dataList* and the output file with the modified data.

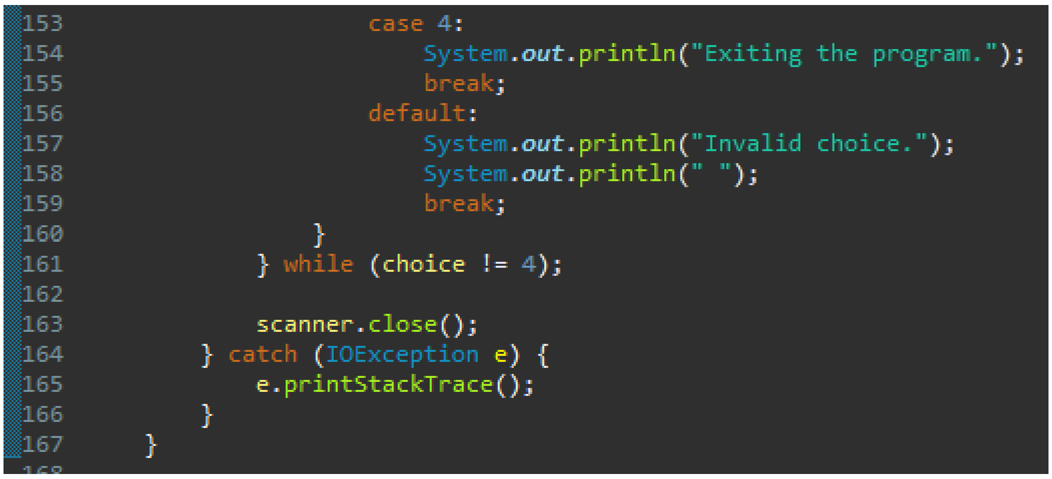


```java
            case 3:
                // Binary search on sorted data based on the Integer part (IDs)
                System.out.print("Enter the target ID to search: ");
                int searchID = scanner.nextInt();
                scanner.nextLine(); // Consume the newline character

                Timer searchTimer = new Timer();
                searchTimer.start();
                ArrayList<Integer> IdList = new ArrayList<>();
                for (int i =0 ;i<formated.size();i+=3)
                {
                    IdList.add(Integer.parseInt(formated.get(i)));
                }
                int index = binarySearch(dataList, searchID);

                if (index != -1) {
                    String foundData = dataList.get(index);
                    System.out.println(foundData);
                    System.out.println(" ");
                } else {
                    System.out.println("ID " + searchID + " not found.");
                    System.out.println(" ");
                }

                searchTimer.stop();
                System.out.println("Search method runtime (excluding user input): " + searchTimer.getElapsedTime() + " nanoseconds");
                System.out.println("Memory Used By Search: " + getMemoryUsedDuringMethod() + " bytes");
                System.out.println(" ");
                break;
```

Inside this case 3 block, the code prompts the user to enter an integer value as the target ID to search for. The code performs a binary search on the sorted data in *dataList* based on the integer part (IDs) to find the target ID. The binary search algorithm efficiently narrows down the search space by dividing the data into halves at each step until the target ID is found or the search space is finished. Before performing the binary search, the code prepares an *ArrayList<Integer>* named *IdList* to hold the integer values of the IDs present in the *formated* list. It extracts the integer part of each entry (ID) from *formated* and adds it to *IdList*. This step is necessary because binary search requires the data to be in a sorted form, and the *formated* list contains a subset of data, which is not necessarily sorted.

Then the code calls the *binarySearch* method, passing the *dataList* and the target ID as parameters. The *binarySearch* method implements the binary search algorithm and returns the index of the target ID if found or -1 if not found. After performing the binary search, the code checks the return value of the *binarySearch* method. If the index is not -1, it means the target ID is found in the *dataList*. The code retrieves the corresponding data from *dataList* and prints it to the console. Otherwise, it informs the user that the target ID was not found.

After that, the code measures the runtime of the "Search" method (excluding user input) using a Timer class. It prints the elapsed time in nanoseconds. And also print the memory space used by the function using *getMemoryUsedDuringMethod*(). This code allows the user to search for a specific ID in the sorted data and provides information on whether the ID is found or not. It uses a binary search algorithm for efficient searching.


```
                        case 4:
                            System.out.println("Exiting the program.");
                            break;
                        default:
                            System.out.println("Invalid choice.");
                            System.out.println(" ");
                            break;
                    }
                } while (choice != 4);

                scanner.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
```

Inside this case 4 block, when the user selects option 4 from the menu, the code prints the message "Exiting the program." to inform the user that the program is terminating.
In this part of code also includes a default case in the switch statement to handle any invalid choices made by the user. If the user enters a choice other than 1, 2, 3, or 4, the default case will be executed. It prints the message "Invalid choice." to inform the user that the input is not recognized as a valid option. After processing the user's choice, the code checks if the choice variable is equal to 4. If it is, the loop will terminate, and the program will exit. Otherwise, the loop will continue to prompt the user for input until they choose to exit (select option 4). After the loop ends, the code closes the Scanner object (scanner) to release any resources associated with it.

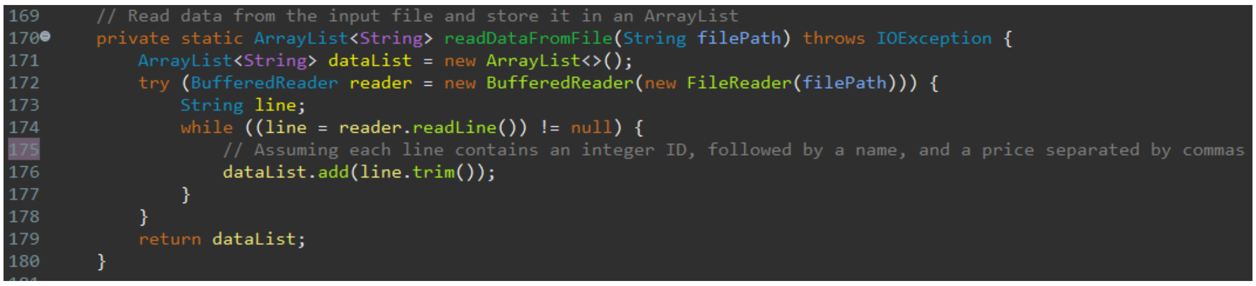
In this part of code, it is enclosed in a try-catch block, where it catches any *IOException* that may occur during file read/write operations or if any other I/O-related error occurs. If such an exception occurs, it prints the exception's stack trace to the console. This allows the program to handle potential errors gracefully instead of crashing abruptly.

<u>Function in main function:</u>

1- **readDataFromFile() :**

This method takes a single argument: *filePath*, which is a string representing the path to the input file. Then it will create an ArrayList named *dataList* to store the data read from the file. The method then uses a try-with-resources block to create a *BufferedReader* that reads data from the specified file. The try-with-resources ensures that the *BufferedReader* is closed properly after the execution of the block. It enters the while loop that reads each line from the file using *reader.readLine()* until there are no more lines to read (*reader.readLine()* returns null). Inside the while loop, each line read from the file is added to the *dataList* after removing any leading or trailing commas using the function *line.trim()*.

When the loop finishes reading all the lines from the database (.txt file), the method closes the BufferedReader and returns the *dataList*, which now contains all the lines of data from the database as strings in the same order they are in the file. The method considers that each line in the file contains data in the format of "integer ID, String name, double price $, integer quantity" all separated by spaces. For example, a first line "0002,Toothpaste,89.12$,14". When we call this method with the file path, it will read the file and return an *ArrayList* of strings. If there are any issues with reading the file or if the file does not follow the expected format, the method may throw an *IOException*.

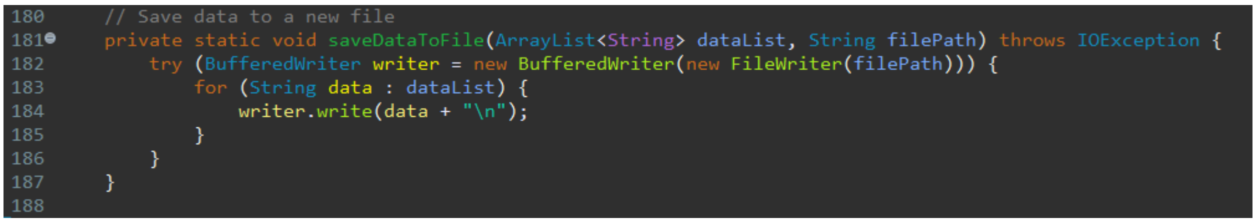


```java
    // Read data from the input file and store it in an ArrayList
    private static ArrayList<String> readDataFromFile(String filePath) throws IOException {
        ArrayList<String> dataList = new ArrayList<>();
        try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
            String line;
            while ((line = reader.readLine()) != null) {
                // Assuming each line contains an integer ID, followed by a name, and a price separated by commas
                dataList.add(line.trim());
            }
        }
        return dataList;
    }
```

2- saveDataToFile():

This method takes two arguments: *dataList*, which is an *ArrayList* of strings containing the data that needs to be saved, and *filePath*, which is a string representing the path to the output file where the data will be saved. The method uses a try-with-resources block to create a *BufferedWriter* that will be used to write data to the specified file. The try-with-resources ensures that the *BufferedWriter* is closed properly after the execution of the block. It enters a for loop that iterates over each element (string) in the *dataList*. Inside the for loop, it writes each element (data) to the *BufferedWriter* by using *writer.write(data + "\n")*. The *"\n"* is added to insert a new line after each data element, so each data element will be written on a new line in the output file.

When the loop finishes writing all the data elements to the file, the method closes the *BufferedWriter*, ensuring that the data is removed and saved to the output file. This *saveDataToFile* method takes an *ArrayList* of strings containing data and writes each element of the list to the output file, with each element written on separate line. If the file already exists, it will be overwritten with the new data. If the file doesn't exist, a new file will be created at the specified file path, and the data will be written to it. The method will throw an *IOException* if there are any issues with writing the data to the file, such as if the file is read-only or if there are permission-related problems.



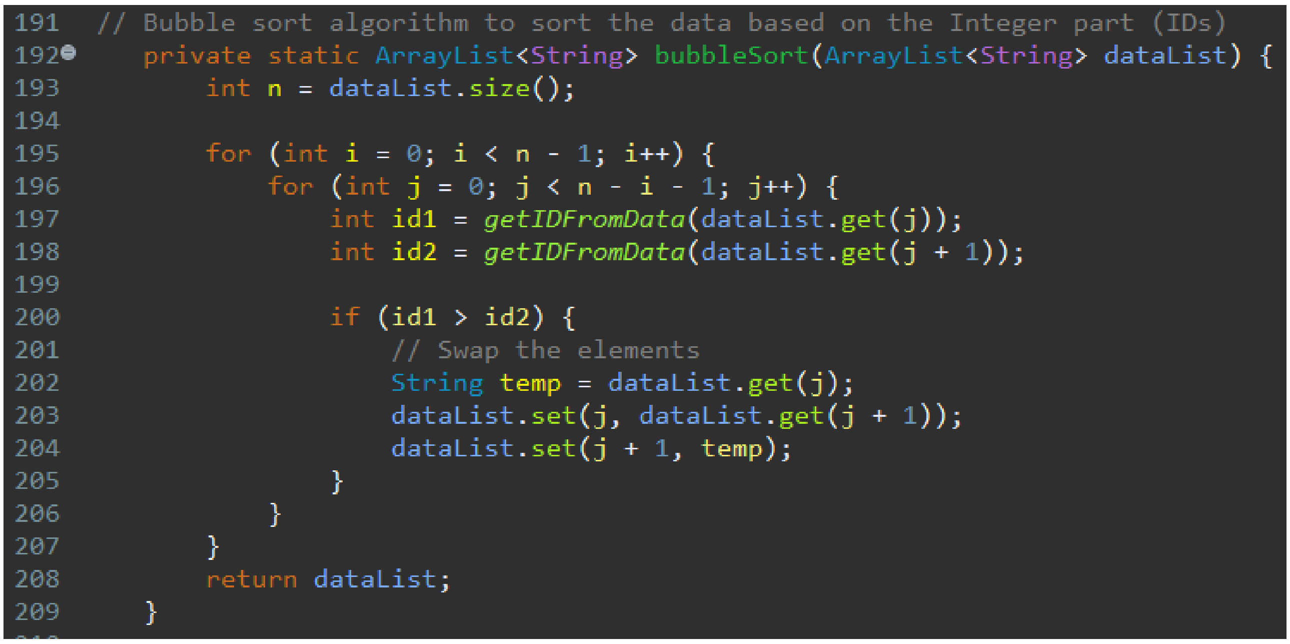
```java
// Save data to a new file
private static void saveDataToFile(ArrayList<String> dataList, String filePath) throws IOException {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(filePath))) {
        for (String data : dataList) {
            writer.write(data + "\n");
        }
    }
}
```

3- **bubbleSort():**

This method takes a single argument *dataList*, which is an *ArrayList* of strings containing the data to be sorted. It calculates the size of the *dataList* and stores it in the variable *n*. It enters two nested for loops. The outer for loop, controlled by the variable *i*, goes from 0 to n-1. The purpose of this loop is to ensure that the largest element "bubbles" to the end of the list in each iteration. The inner for loop, controlled by the variable *j*, runs from 0 to n-i-1. The purpose of this for loop is to compare adjacent elements and swap them if they are in the wrong order (unsorted).

Inside the inner for loop, the method extracts the Integer IDs from two adjacent elements using the *getIDFromData* method. It does this for the j-th element and the j+1-th element. It compares the two IDs, id1 and id2, and if id1 is greater than id2, it means that the elements are in the wrong order based on their IDs (unsorted). In this a case, the method will swap between the elements using a temporary variable temp. This way, the larger ID will "bubble up" towards the end of the list. The inner for loop continues until all adjacent elements have been compared and swapped, also for the current value of *i*, if necessary.

When the outer for loop finishes, the list will be sorted based on the Integer IDs in ascending order. At the end, the method returns the sorted *dataList* as the final output. The method modifies the original *dataList*, and after it completes, the list will contain the same data but sorted based on the Integer part of each element (ID).



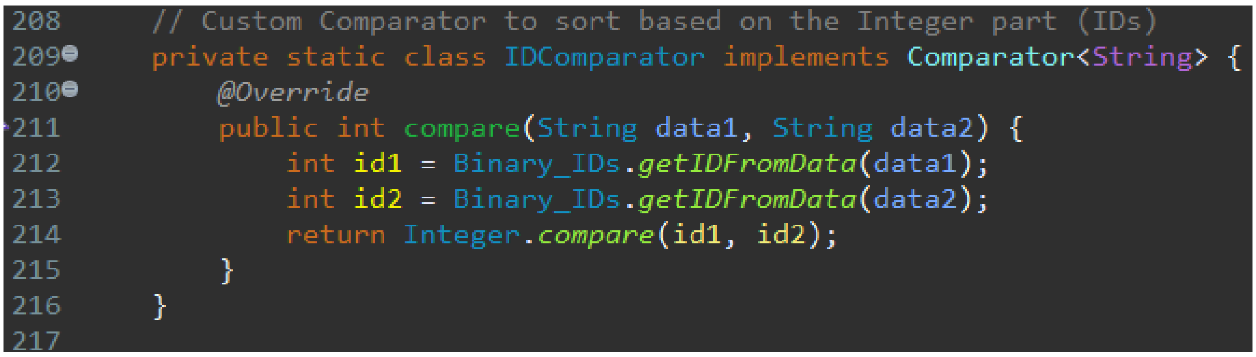
```java
// Bubble sort algorithm to sort the data based on the Integer part (IDs)
    private static ArrayList<String> bubbleSort(ArrayList<String> dataList) {
        int n = dataList.size();

        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                int id1 = getIDFromData(dataList.get(j));
                int id2 = getIDFromData(dataList.get(j + 1));

                if (id1 > id2) {
                    // Swap the elements
                    String temp = dataList.get(j);
                    dataList.set(j, dataList.get(j + 1));
                    dataList.set(j + 1, temp);
                }
            }
        }
        return dataList;
    }
```

4- **compare() :**

This method is a nested static class, which means it is a class defined within another class (*Binary_IDs*). Being static means that you can use it without creating an instance of the outer class. It implements the compare method of the Comparator interface, which is used to compare two string elements, *data1* and *data2*, and determine their relative ordering. Inside the compare method, it calls the *getIDFromData* method, which is a static method defined in the *Binary_IDs* class (the parent class), to extract the Integer ID from each string element (*data1* and *data2*). It then uses *Integer.compare(id1, id2)* to compare the extracted IDs (*id1* and *id2*). The *Integer.compare()* method returns an integer value that indicates the relationship between the two IDs:

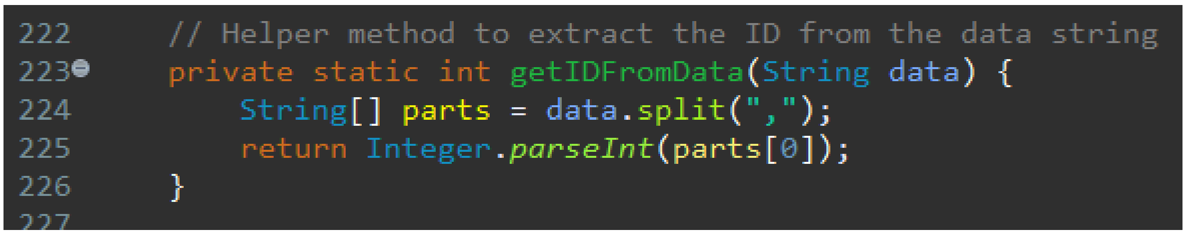
- If *id1* is less than *id2*, it returns a negative value.
- If *id1* is equal to *id2*, it returns 0.
- If *id1* is greater than *id2*, it returns a positive value.

This comparison result will determine the order of *data1* and *data2* in the sorted list. If the result is negative, *data1* will come before *data2*. If the result is positive, *data1* will come after *data2*. If the result is 0, the order of *data1* and *data2* will be considered equal in terms of sorting. By defining this custom comparator, we can now use it to sort lists of strings based on their Integer IDs. For example, we can use Collections.sort(*dataList*, *new IDComparator()*) to sort the *dataList* in ascending order based on the Integer part of each string element.


```
    // Custom Comparator to sort based on the Integer part (IDs)
    private static class IDComparator implements Comparator<String> {
        @Override
        public int compare(String data1, String data2) {
            int id1 = Binary_IDs.getIDFromData(data1);
            int id2 = Binary_IDs.getIDFromData(data2);
            return Integer.compare(id1, id2);
        }
    }
```

5- getIDFromData() :

This method takes a single argument data, which is a string containing the data from which we want to extract the ID. It uses the *split()* method of the String class to split the data string into an array of strings based on the space character *(",")*. The *split(",")* call separates the data string wherever it finds a space and stores the substrings into an array. The extracted substrings are stored in the parts array. The method then converts the first element of the parts array (ID) to an integer using *Integer.parseInt(parts[0])* and returns it as the result. For example, if the input data string is " 0001,Shampoo,55.79$", the *getIDFromData* method will extract "0001" as the ID and convert it to the integer 1, which will be returned. The method assumes that the input data string is formatted in such a way that the ID is the first part and is followed by other data separated by spaces. If the data string does not follow this specific format, the method will throw an exception. Therefore, it's essential to ensure that the data string provided to this method adheres to the expected format.



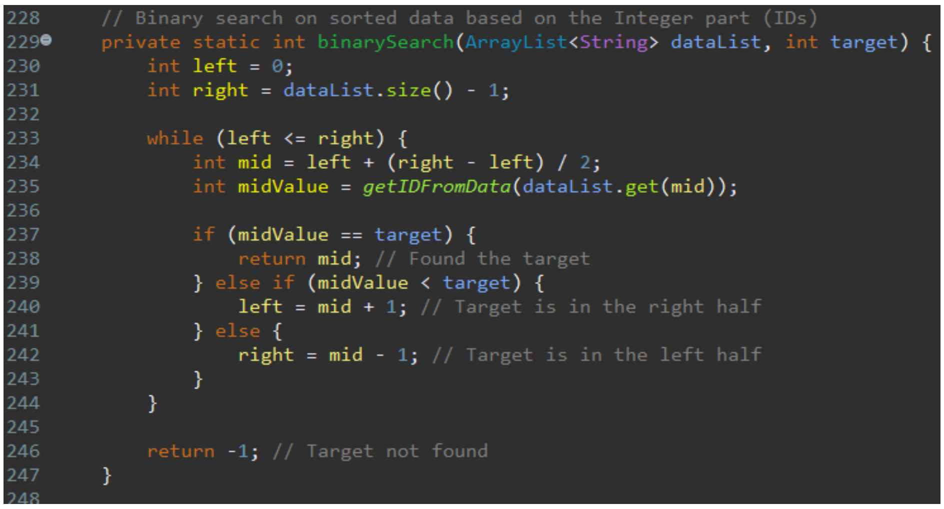
```java
    // Helper method to extract the ID from the data string
    private static int getIDFromData(String data) {
        String[] parts = data.split(",");
        return Integer.parseInt(parts[0]);
    }
```

6- binarySearch() :

This method takes two arguments: *dataList*, which is an ArrayList of strings containing the sorted data, and target, which is the integer value that we want to find in the list. It initializes two variables, left and right, representing the left and right boundaries of the search range in the list. left is initialized to 0 (the first index of the list), and right is initialized to the last index of the list, which is *dataList.size()* - 1. It enters a while loop that continues if the left boundary is less than or equal to the right boundary. If the left boundary becomes greater than the right boundary, it means the target value is not present in the list, and the while loop will terminate.

Inside the while loop, it calculates the middle index mid as the average of left and right, rounded down to the nearest integer, to avoid overflow issues. It extracts the Integer ID from the element at the mid index of the *dataList* using the *getIDFromData* method and stores it in the variable *midValue*. It compares *midValue* with the target value. If they are equal, it means the target has been found at the middle index mid, and the method returns mid. If *midValue* is less than the target, it means the target value is in the right half of the current search range. In this case, the left boundary is updated to mid + 1, effectively eliminating the left half of the current range from consideration. If *midValue* is greater than the target, it means the target value is in the left half of the current search range. In this case, the right boundary is updated to mid - 1, effectively eliminating the right half of the current range from consideration.

The loop continues to narrow down the search range by updating left and right until the target value is found (and returned as the index of the target) or until the left boundary becomes greater than the right boundary, indicating that the target value is not present in the list. If the target value is not found, the method returns -1, indicating that the target is not present in the list.

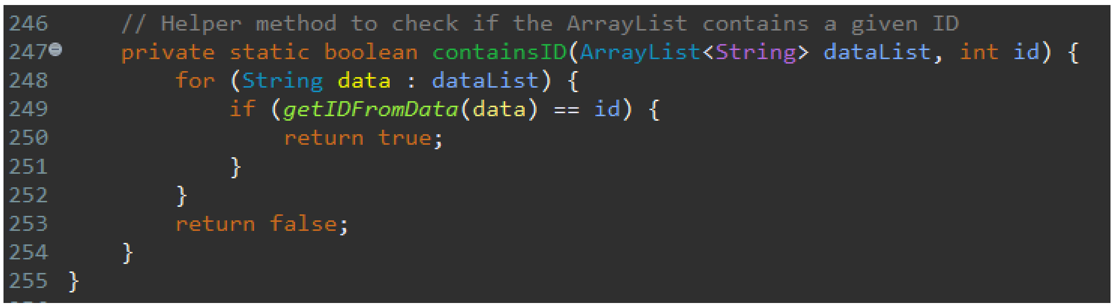


```
    // Binary search on sorted data based on the Integer part (IDs)
    private static int binarySearch(ArrayList<String> dataList, int target) {
        int left = 0;
        int right = dataList.size() - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;
            int midValue = getIDFromData(dataList.get(mid));

            if (midValue == target) {
                return mid; // Found the target
            } else if (midValue < target) {
                left = mid + 1; // Target is in the right half
            } else {
                right = mid - 1; // Target is in the left half
            }
        }

        return -1; // Target not found
    }
```

7- **containsID():**

This method takes two arguments: *dataList*, which is an ArrayList of strings containing the data to be searched, and id, which is the integer ID that we want to check if it exists in the list. It enters a for-each loop that iterates through each element (string) in the *dataList*. The for loop variable data represents the current element being processed in each iteration. Inside the for loop, it calls the *getIDFromData* method to extract the integer ID from the data string. It compares the extracted id with the given id. If they are equal, it means the given ID is found in the *dataList*, and the method returns true.

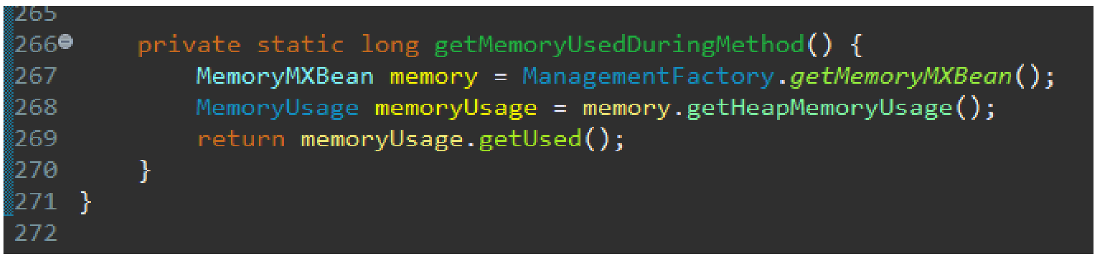
The for loop continues to iterate through the *dataList*, checking each element's ID until either the ID is found, and the method returns true, or all elements in the *dataList* have been checked. If the loop finishes iterating through the entire *dataList* without finding the given id, the method returns false, indicating that the given ID is not present in the list.



```java
    // Helper method to check if the ArrayList contains a given ID
    private static boolean containsID(ArrayList<String> dataList, int id) {
        for (String data : dataList) {
            if (getIDFromData(data) == id) {
                return true;
            }
        }
        return false;
    }
}
```

8- getMemoryUsedDuringMethod():

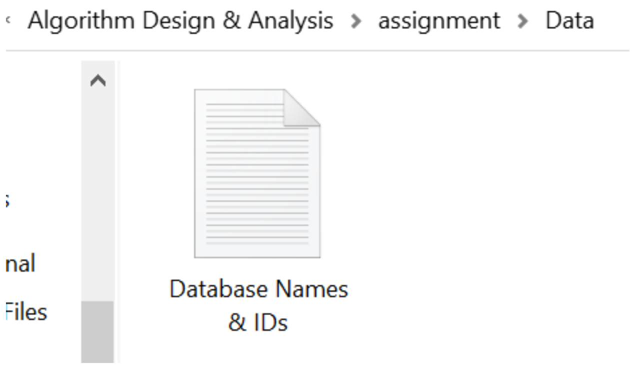
This method calculates the amount of memory used by accessing the *MemoryMXBean*, which provides memory usage information. Specifically, it retrieves the heap memory usage using *getHeapMemoryUsage*() and returns the number of bytes of heap memory currently in use with *memoryUsage.getUsed*(). This functionality allows us to monitor and analyze memory consumption during the execution of main methods, aiding in memory profiling and optimization. In conclusion, this method allows us to monitor and retrieve the amount of heap memory used at a specific point in our Java code. It's often useful for memory profiling and optimization to understand memory consumption during program execution.

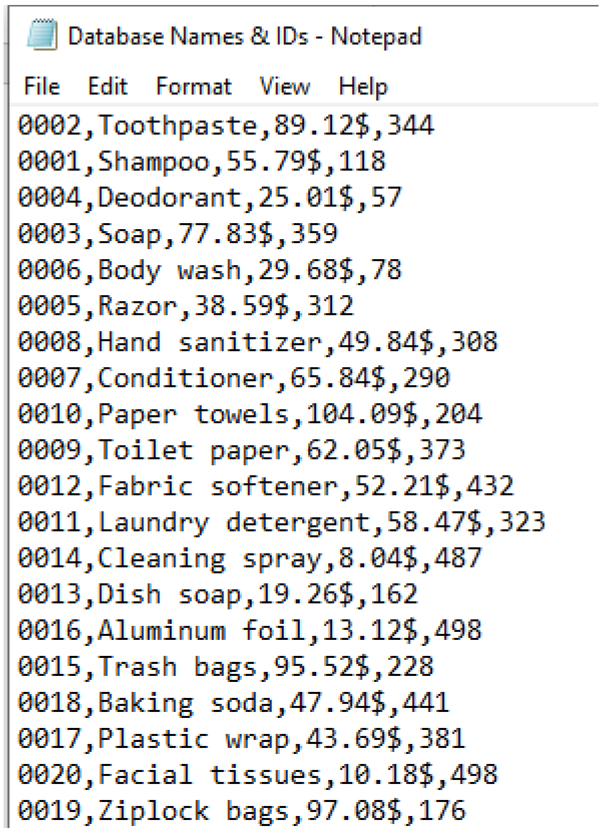


```java
    private static long getMemoryUsedDuringMethod() {
        MemoryMXBean memory = ManagementFactory.getMemoryMXBean();
        MemoryUsage memoryUsage = memory.getHeapMemoryUsage();
        return memoryUsage.getUsed();
    }
}
```

Functionality of the code:

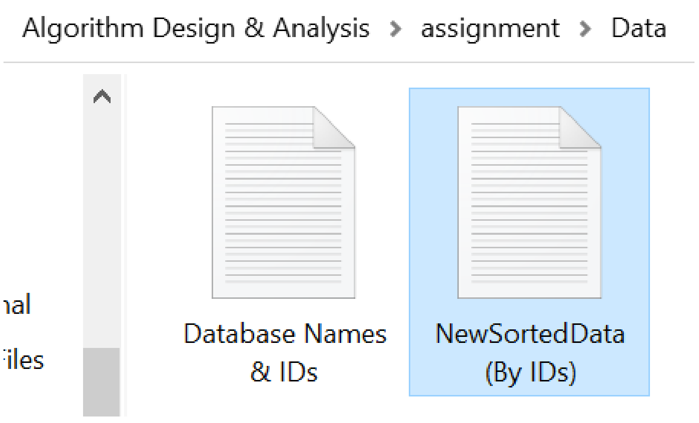
At the beginning, we have a file that contains the database (input path) and the program will go to that path to get the database called "*Database Names & IDs*" (unsorted by Names and IDs)

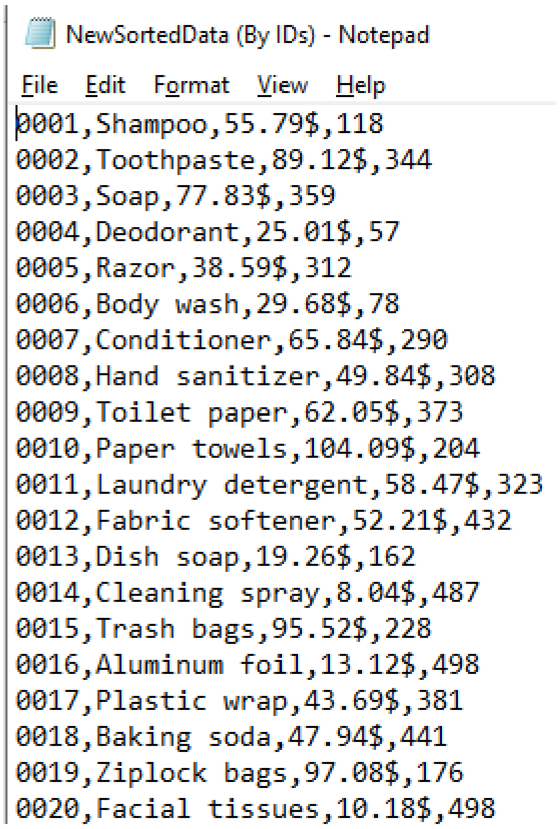




```
0002,Toothpaste,89.12$,344
0001,Shampoo,55.79$,118
0004,Deodorant,25.01$,57
0003,Soap,77.83$,359
0006,Body wash,29.68$,78
0005,Razor,38.59$,312
0008,Hand sanitizer,49.84$,308
0007,Conditioner,65.84$,290
0010,Paper towels,104.09$,204
0009,Toilet paper,62.05$,373
0012,Fabric softener,52.21$,432
0011,Laundry detergent,58.47$,323
0014,Cleaning spray,8.04$,487
0013,Dish soap,19.26$,162
0016,Aluminum foil,13.12$,498
0015,Trash bags,95.52$,228
0018,Baking soda,47.94$,441
0017,Plastic wrap,43.69$,381
0020,Facial tissues,10.18$,498
0019,Ziplock bags,97.08$,176
```

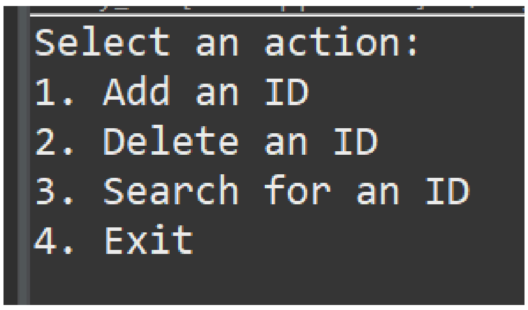
When we start to run the program. It automatically starts by creating a new .txt file (output file) called "*NewSortedData (By IDs)*" that sorts all the data inside it according to their ID number using the bubble sort we explained above.





```
0001,Shampoo,55.79$,118
0002,Toothpaste,89.12$,344
0003,Soap,77.83$,359
0004,Deodorant,25.01$,57
0005,Razor,38.59$,312
0006,Body wash,29.68$,78
0007,Conditioner,65.84$,290
0008,Hand sanitizer,49.84$,308
0009,Toilet paper,62.05$,373
0010,Paper towels,104.09$,204
0011,Laundry detergent,58.47$,323
0012,Fabric softener,52.21$,432
0013,Dish soap,19.26$,162
0014,Cleaning spray,8.04$,487
0015,Trash bags,95.52$,228
0016,Aluminum foil,13.12$,498
0017,Plastic wrap,43.69$,381
0018,Baking soda,47.94$,441
0019,Ziplock bags,97.08$,176
0020,Facial tissues,10.18$,498
```

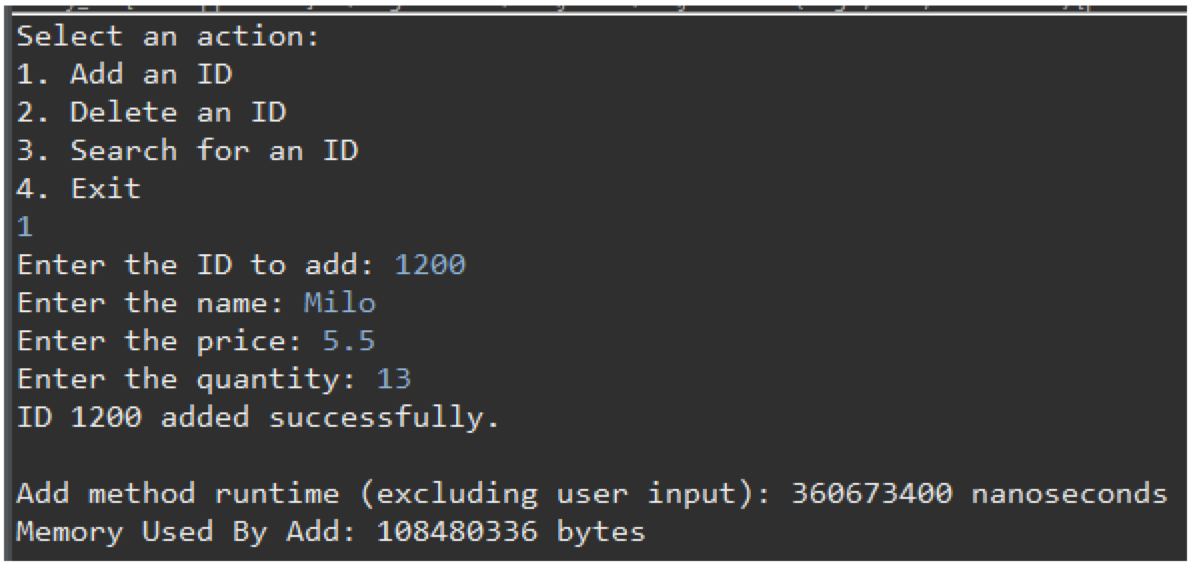
All the work starts in the console. This message is printed in the console, where the user must interact with the program to choose the command he wants:



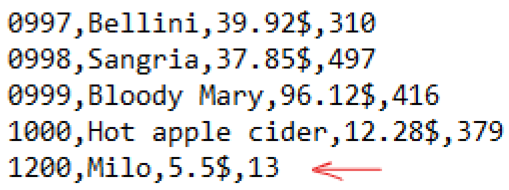
```
Select an action:
1. Add an ID
2. Delete an ID
3. Search for an ID
4. Exit
```

We take the commands one by one. We start by the first one:

1- Add and ID:

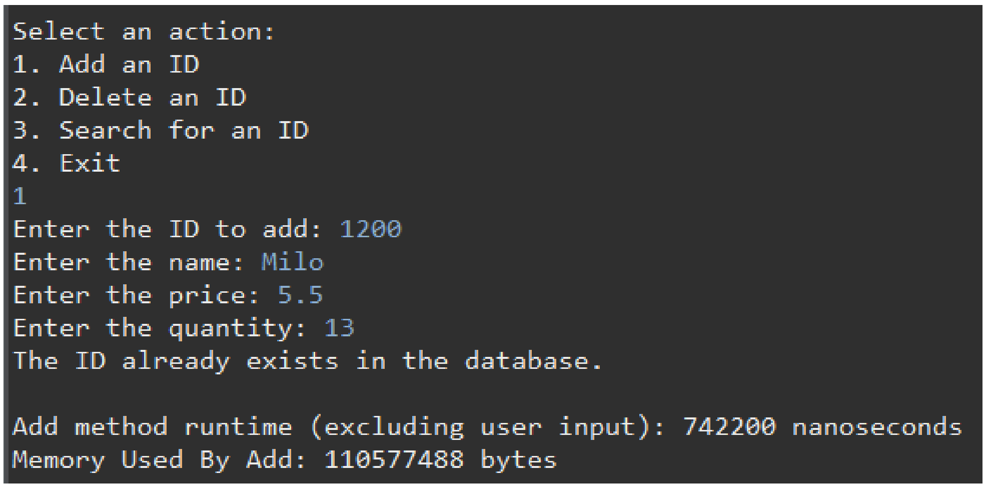


```
Select an action:
1. Add an ID
2. Delete an ID
3. Search for an ID
4. Exit
1
Enter the ID to add: 1200
Enter the name: Milo
Enter the price: 5.5
Enter the quantity: 13
ID 1200 added successfully.

Add method runtime (excluding user input): 360673400 nanoseconds
Memory Used By Add: 108480336 bytes
```



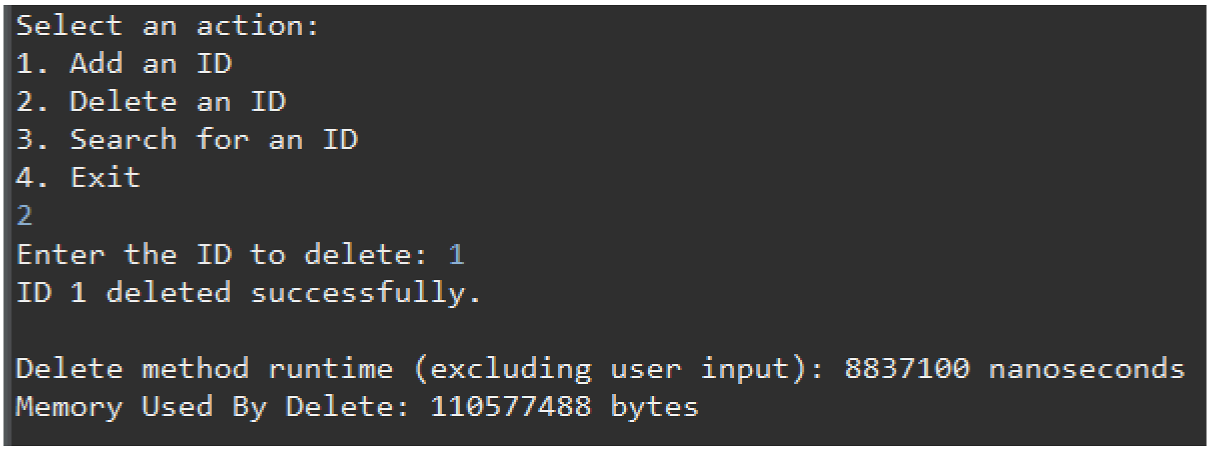
```
0997,Bellini,39.92$,310
0998,Sangria,37.85$,497
0999,Bloody Mary,96.12$,416
1000,Hot apple cider,12.28$,379
1200,Milo,5.5$,13
```

When the user chooses to add, the program asks him to enter the ID number he wants to add. Then it asks him to write the name of the product. And finally, the price of the product. The program will first check if this ID number already exists in the database or not. If it doesn't exist, it will show that the ID number is added successfully and will print the runtime of the method in nanoseconds (excluding the user time) and the memory used by the method. And it will automatically update the output file "*NewSortedData (By IDs)*" If the ID number already exists, the program will print that the ID number already exists in the database. And it will also print the runtime and the memory used.

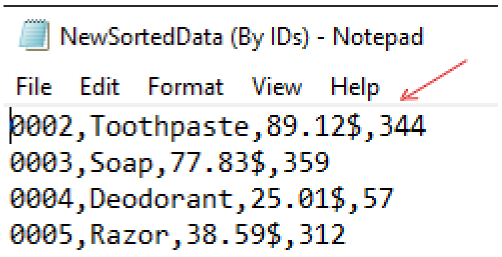


```
Select an action:
1. Add an ID
2. Delete an ID
3. Search for an ID
4. Exit
1
Enter the ID to add: 1200
Enter the name: Milo
Enter the price: 5.5
Enter the quantity: 13
The ID already exists in the database.

Add method runtime (excluding user input): 742200 nanoseconds
Memory Used By Add: 110577488 bytes
```

2- Delete an ID:

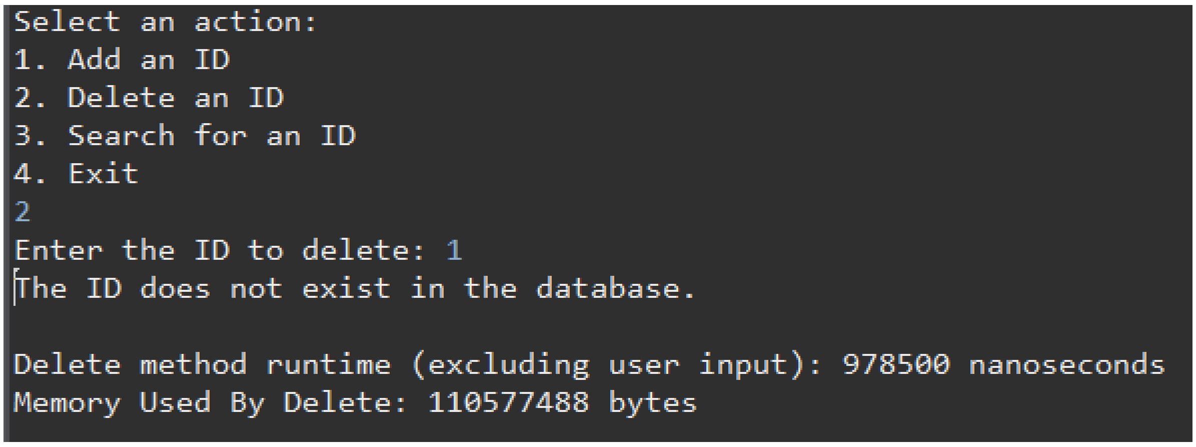


```
Select an action:
1. Add an ID
2. Delete an ID
3. Search for an ID
4. Exit
2
Enter the ID to delete: 1
ID 1 deleted successfully.

Delete method runtime (excluding user input): 8837100 nanoseconds
Memory Used By Delete: 110577488 bytes
```

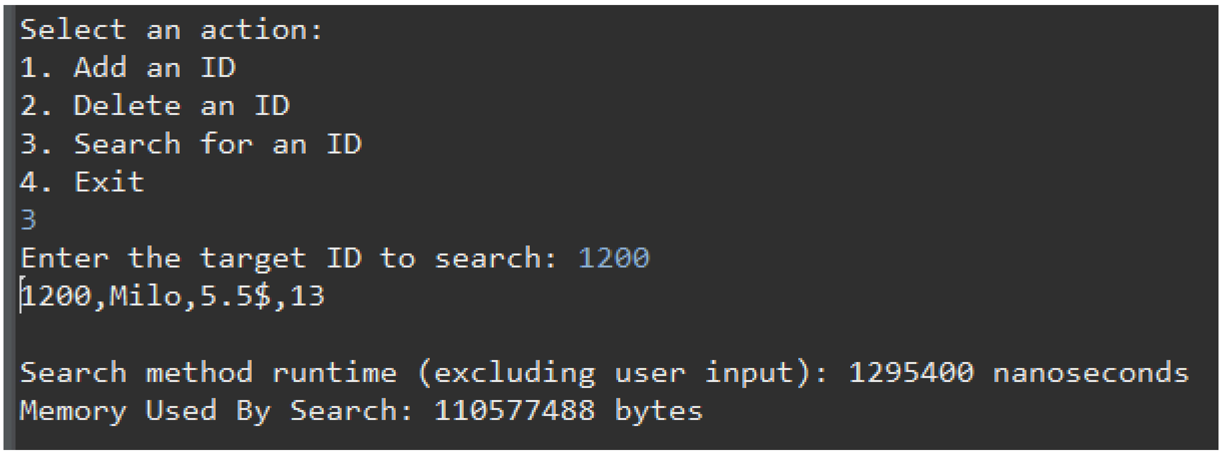


```
0002,Toothpaste,89.12$,344
0003,Soap,77.83$,359
0004,Deodorant,25.01$,57
0005,Razor,38.59$,312
```

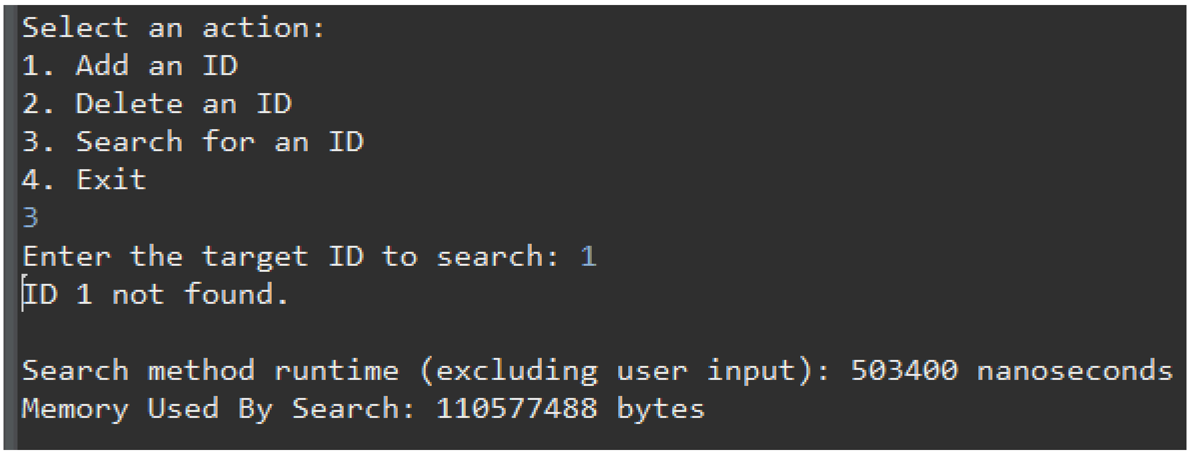
When the user chooses to delete, the program asks him to enter the ID number he wants to delete. The program will first check if this ID number already exists in the database or not. If the ID number exists in the database, it will perform the delete action and remove it from the database and will print the runtime of the method in nanoseconds (excluding the user time) and the memory used by the method. And it will automatically update the output file "*NewSortedData (By IDs)*". If it doesn't exist, it will show that the ID number does not exist in the database. And it will also print the runtime and the memory used.


```
Select an action:
1. Add an ID
2. Delete an ID
3. Search for an ID
4. Exit
2
Enter the ID to delete: 1
The ID does not exist in the database.

Delete method runtime (excluding user input): 978500 nanoseconds
Memory Used By Delete: 110577488 bytes
```

3- Search for an ID:

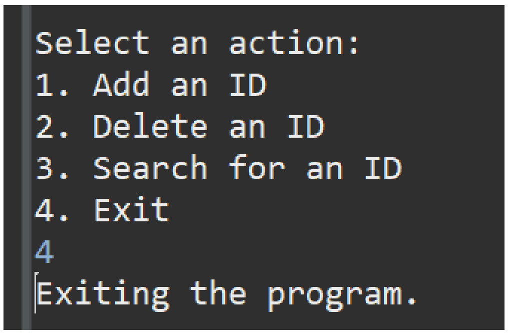

```
Select an action:
1. Add an ID
2. Delete an ID
3. Search for an ID
4. Exit
3
Enter the target ID to search: 1200
1200,Milo,5.5$,13

Search method runtime (excluding user input): 1295400 nanoseconds
Memory Used By Search: 110577488 bytes
```

When the user chooses to search, the program asks him to enter the ID number he wants to search for. The program will first check if this ID number already exists in the database or not. If the ID number exists in the database, it will print all the data related to this ID number from the database (ID, Name, Price, Quantity) and will print the runtime of the method in nanoseconds (excluding the user time) and the memory used by the method. And it will automatically update the output file "*NewSortedData (By IDs)*". If it doesn't exist, it will show that the ID number is not found in the database. And it will also print the runtime and the memory used.



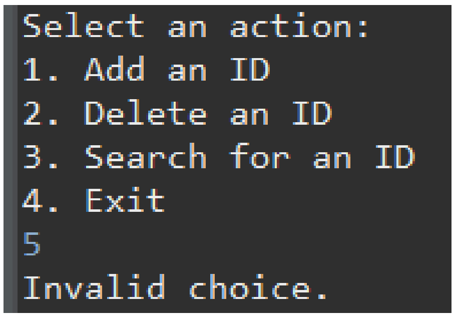
```
Select an action:
1. Add an ID
2. Delete an ID
3. Search for an ID
4. Exit
3
Enter the target ID to search: 1
ID 1 not found.

Search method runtime (excluding user input): 503400 nanoseconds
Memory Used By Search: 110577488 bytes
```

4- Exit:



```
Select an action:
1. Add an ID
2. Delete an ID
3. Search for an ID
4. Exit
4
Exiting the program.
```

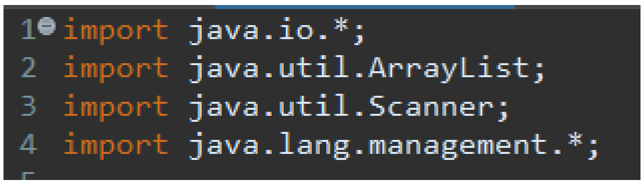
When the user chooses to exit. The program terminates everything and stops running. and it leaves the output file. If you enter any other option that is not in the option, you get a message saying: Invalid choice. And get you back to the options of action.



```
Select an action:
1. Add an ID
2. Delete an ID
3. Search for an ID
4. Exit
5
Invalid choice.
```

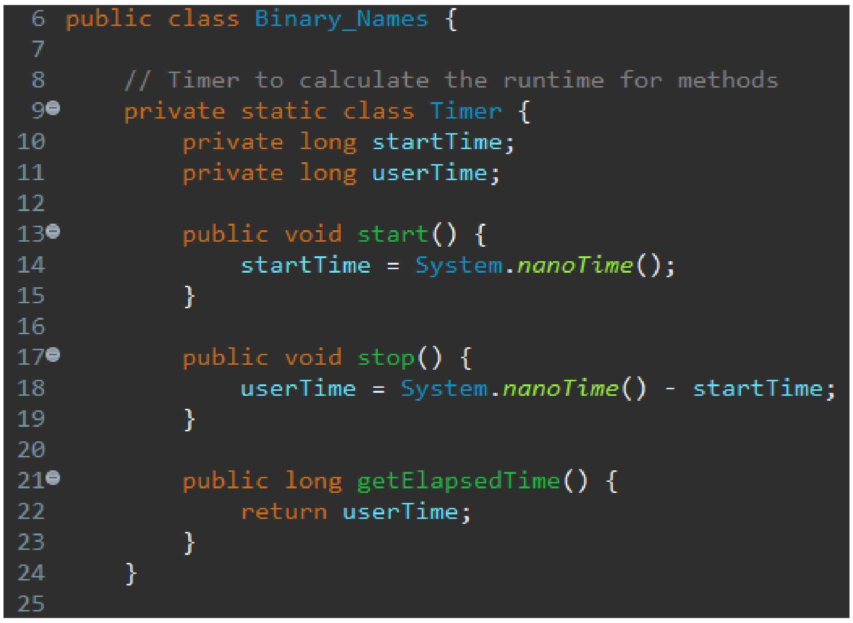
II- Product NAMES Application 1: Arraylist + binary search + Bubble sort:

We will start by explaining the code then show the program running. We will divide the code into different parts so it will be easy to explain and to understand.



```java
import java.io.*;
import java.util.ArrayList;
import java.util.Scanner;
import java.lang.management.*;
```

We start by importing all the packages, sub-packages, classes, interfaces we need for the code.



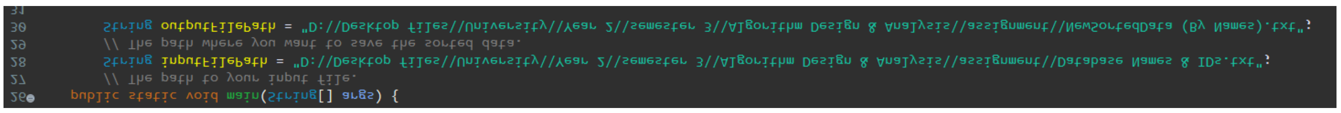
```java
public class Binary_Names {

    // Timer to calculate the runtime for methods
    private static class Timer {
        private long startTime;
        private long userTime;

        public void start() {
            startTime = System.nanoTime();
        }

        public void stop() {
            userTime = System.nanoTime() - startTime;
        }

        public long getElapsedTime() {
            return userTime;
        }
    }

```

So, the name of the class is "*Binary_Names*"

We started by making a timer class that calculates the run time of the program without considering the time taken by the user to enter data. Along the explanation of the code, we will find that we have used this timer method to calculate the runtime of each task the user chooses to execute (search, add, delete).

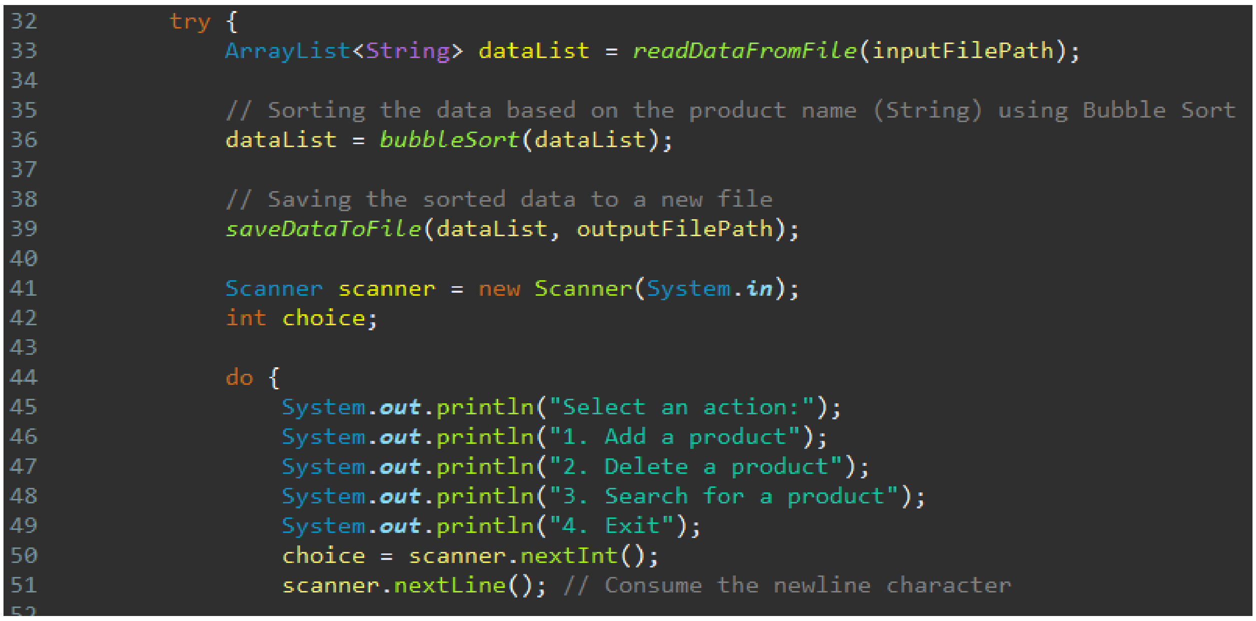
Inside this timer class, we have 3 methods:

1. *start()* : to start the time
2. *stop()* : to calculate the final time after it removes the time the user takes to put the input
3. *getElapsedTime()* : it returns the runtime



```java
    public static void main(String[] args) {
        // The path to your input file.
        String inputFilePath = "D://Desktop files//University//Year 2//semester 3//Algorithm Design & Analysis//assignment//Database Names & IDs.txt";
        // The path where you want to save the sorted data.
        String outputFilePath = "D://Desktop files//University//Year 2//semester 3//Algorithm Design & Analysis//assignment//NewSortedData (By Names).txt";
```

Here, we started writing in the main function (the one that program runs). We indicated the path (location in your PC) for database (input file) and the path for the new updated or edited database (output file).

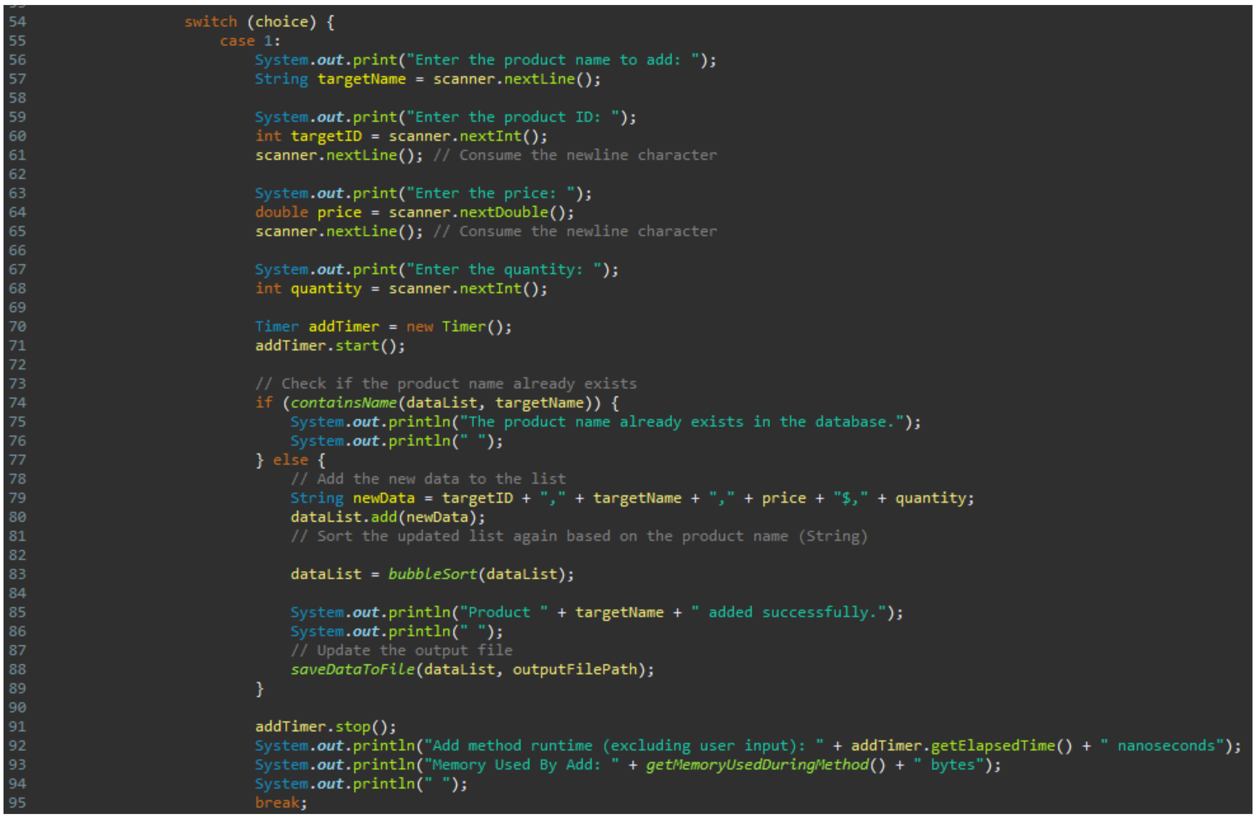


```
        try {
            ArrayList<String> dataList = readDataFromFile(inputFilePath);

            // Sorting the data based on the product name (String) using Bubble Sort
            dataList = bubbleSort(dataList);

            // Saving the sorted data to a new file
            saveDataToFile(dataList, outputFilePath);

            Scanner scanner = new Scanner(System.in);
            int choice;

            do {
                System.out.println("Select an action:");
                System.out.println("1. Add a product");
                System.out.println("2. Delete a product");
                System.out.println("3. Search for a product");
                System.out.println("4. Exit");
                choice = scanner.nextInt();
                scanner.nextLine(); // Consume the newline character
```

In this part of code, the code reads data from a file specified by the variable *inputFilePath* and stores it in an *ArrayList* named *dataList*. Then the data in *dataList* is firstly sorted based on the integer part (IDs) using a custom comparator called *IDComparator*. The *IDComparator* class implements the Comparator interface to define a custom sorting order for the data. After sorting, the code extracts the first three elements from each string in the *dataList*, separated by commas *(,)*. These elements are then added to a new *ArrayList* named *formated*. Next, the sorted data in *dataList* is saved to a new file specified by the variable *outputFilePath* using the *saveDataToFile* method. After the data is sorted and saved, the code creates a Scanner object to interact with the user. It presents a menu to the user with four options:

- Option 1: Add an ID
- Option 2: Delete an ID
- Option 3: Search for an ID
- Option 4: Exit

The code enters a do-while loop to continuously prompt the user for input until they choose to exit (select option 4). Depending on the user's choice (1, 2, or 3), additional actions could be implemented to add, delete, or search for an ID in the data.

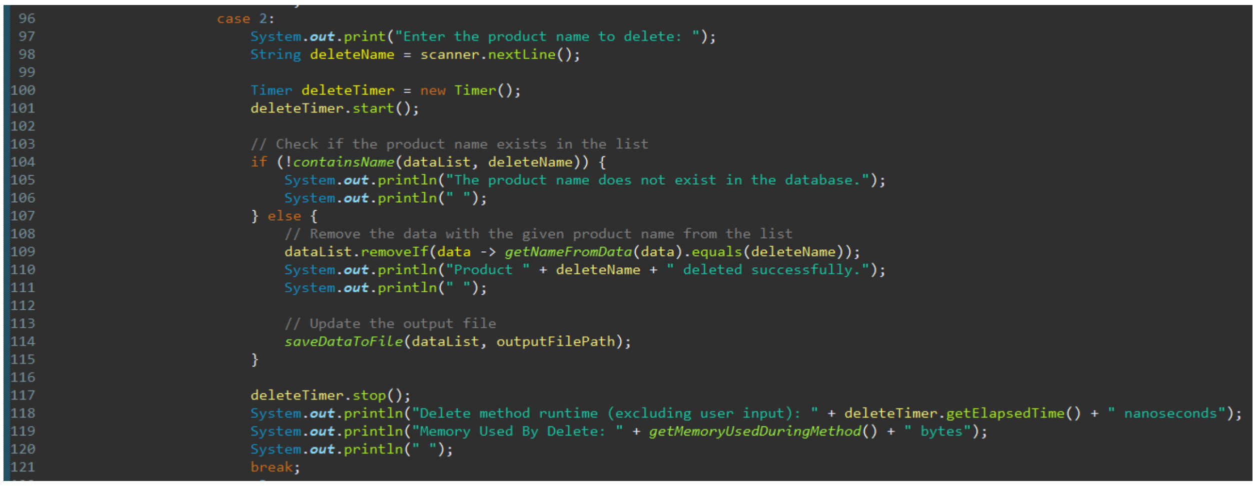


```java
switch (choice) {
    case 1:
        System.out.print("Enter the product name to add: ");
        String targetName = scanner.nextLine();

        System.out.print("Enter the product ID: ");
        int targetID = scanner.nextInt();
        scanner.nextLine(); // Consume the newline character

        System.out.print("Enter the price: ");
        double price = scanner.nextDouble();
        scanner.nextLine(); // Consume the newline character

        System.out.print("Enter the quantity: ");
        int quantity = scanner.nextInt();

        Timer addTimer = new Timer();
        addTimer.start();

        // Check if the product name already exists
        if (containsName(dataList, targetName)) {
            System.out.println("The product name already exists in the database.");
            System.out.println(" ");
        } else {
            // Add the new data to the list
            String newData = targetID + "," + targetName + "," + price + "$," + quantity;
            dataList.add(newData);
            // Sort the updated list again based on the product name (String)

            dataList = bubbleSort(dataList);

            System.out.println("Product " + targetName + " added successfully.");
            System.out.println(" ");
            // Update the output file
            saveDataToFile(dataList, outputFilePath);
        }

        addTimer.stop();
        System.out.println("Add method runtime (excluding user input): " + addTimer.getElapsedTime() + " nanoseconds");
        System.out.println("Memory Used By Add: " + getMemoryUsedDuringMethod() + " bytes");
        System.out.println(" ");
        break;
```

Inside this case 1 block, the code prompts the user to enter the following information:

- A string for the product name to add.
- An integer value for the product ID.
- A double value for the product price.

In the code, we check if the entered product name already exists in the *dataList* (which contains the sorted data read from the file earlier). It does this by calling the *containsName* method. Its target is to prevent adding duplicate product names to the list. If the entered product name does not exist in the *dataList*, the code proceeds to add the new data to the list. It creates a new string *newData* that combines the entered product ID, product name, and product price in the format: "ID,name,price$,quantity". For example, "890,Juice,25.5$,45". After we add the new data, the code uses the *bubbleSort*() method to sort the updated *dataList* again based on the product name (String). The *bubbleSort*() method is called to rearrange the list in alphabetical order based on the product names. Then, the updated and sorted *dataList* is saved to the output file specified by *outputFilePath* using the *saveDataToFile* method. This ensures that the changes are reflected in the file. This code also measures the runtime of the "Add" method (excluding user input) using a Timer class. It prints the elapsed time in nanoseconds. And also print the memory space used by the function using *getMemoryUsedDuringMethod*(). This code allows the user to add a new product to the data list, checks for duplicate product names, sorts the list based on product names, saves the updated list to the output file, and provides runtime and memory used too.

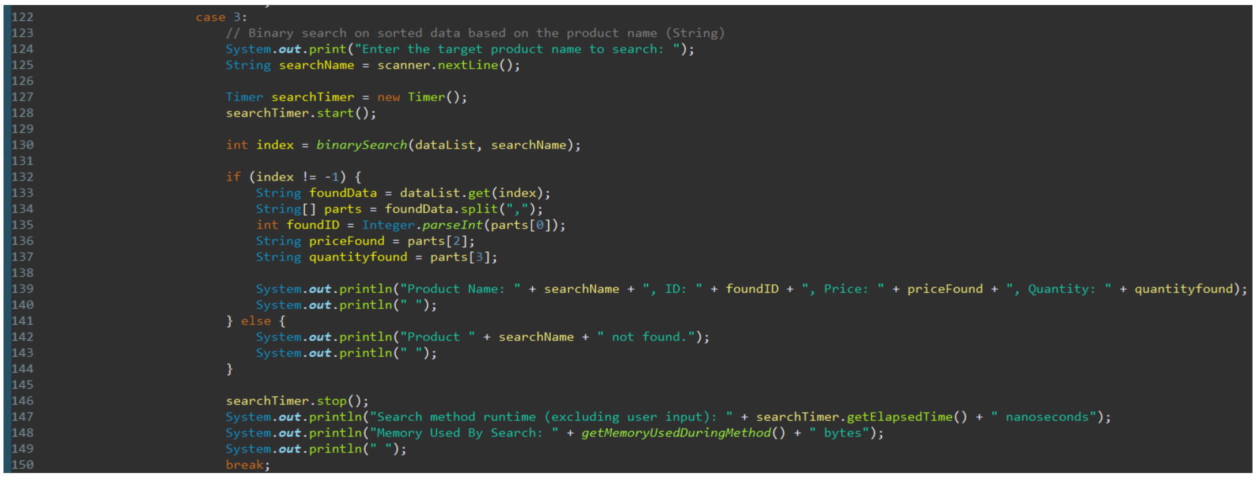


```
case 2:
    System.out.print("Enter the product name to delete: ");
    String deleteName = scanner.nextLine();

    Timer deleteTimer = new Timer();
    deleteTimer.start();

    // Check if the product name exists in the list
    if (!containsName(dataList, deleteName)) {
        System.out.println("The product name does not exist in the database.");
        System.out.println(" ");
    } else {
        // Remove the data with the given product name from the list
        dataList.removeIf(data -> getNameFromData(data).equals(deleteName));
        System.out.println("Product " + deleteName + " deleted successfully.");
        System.out.println(" ");

        // Update the output file
        saveDataToFile(dataList, outputFilePath);
    }

    deleteTimer.stop();
    System.out.println("Delete method runtime (excluding user input): " + deleteTimer.getElapsedTime() + " nanoseconds");
    System.out.println("Memory Used By Delete: " + getMemoryUsedDuringMethod() + " bytes");
    System.out.println(" ");
    break;
```

Inside this case 2 block, this code is part of the switch statement that handles the user's choice for "Delete" operation in the menu. The user is prompted to enter the name of the product they want to delete. A timer named *deleteTimer* is created and started using the *start()* method. This timer will be used to measure the runtime of the delete method (excluding the time taken by the user to enter data). The program checks whether the product name entered by the user exists in the *dataList*. This check is done using the *containsName* method. If the product name does not exist in the database (*containsName(dataList, deleteName)* returns *false*), a message is printed saying that the product name does not exist, and the program moves on to the next step. If the product name exists in the database (*containsName(dataList, deleteName)* returns *true*), the program proceeds to remove the data associated with that product name from the *dataList*.

It uses the *removeIf* method to delete the data that matches the input product name. The method *getNameFromData(data)* is to extract the product name from the data object. A success message is printed indicating that the product with the given name has been deleted successfully. The *dataList* is updated and saved to the output file specified by *outputFilePath* using the *saveDataToFile* method. The *deleteTimer* is stopped using the *stop()* method. The program prints the runtime of the delete method (excluding user input) by calling *deleteTimer.getElapsedTime()*, which presumably returns the time since the timer was started. And also print the memory space used by the function using *getMemoryUsedDuringMethod*(). The break statement is used to exit the loop or switch case block. This indicates that the process of deleting the product is complete.

This code allows the user to delete a product by providing its name from the *dataList*, and then it updates the database with the updated list of products after the deletion. The runtime of the delete method is measured using a timer, which gives an indication of how long it takes to delete the product from the list and how much memory it has used.

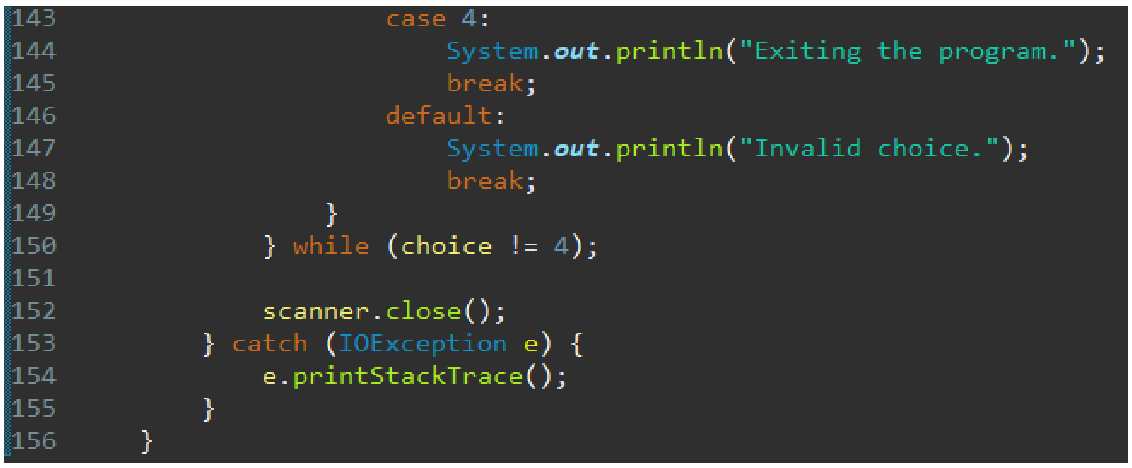


```java
        case 3:
            // Binary search on sorted data based on the product name (String)
            System.out.print("Enter the target product name to search: ");
            String searchName = scanner.nextLine();

            Timer searchTimer = new Timer();
            searchTimer.start();

            int index = binarySearch(dataList, searchName);

            if (index != -1) {
                String foundData = dataList.get(index);
                String[] parts = foundData.split(",");
                int foundID = Integer.parseInt(parts[0]);
                String priceFound = parts[2];
                String quantityfound = parts[3];

                System.out.println("Product Name: " + searchName + ", ID: " + foundID + ", Price: " + priceFound + ", Quantity: " + quantityfound);
                System.out.println(" ");
            } else {
                System.out.println("Product " + searchName + " not found.");
                System.out.println(" ");
            }

            searchTimer.stop();
            System.out.println("Search method runtime (excluding user input): " + searchTimer.getElapsedTime() + " nanoseconds");
            System.out.println("Memory Used By Search: " + getMemoryUsedDuringMethod() + " bytes");
            System.out.println(" ");
            break;
```

Inside this case 3 block, the code prompts the user to enter the product name they want to search for. Then the code will perform a binary search on the sorted data in *dataList* based on the product name (String) to find the target product name. The binary search algorithm efficiently narrows down the search space by dividing the data into halves at each step until the target product name is found or the search space is finished.

After that, the code calls the *binarySearch* method, passing the *dataList* and the target product name as parameters. The *binarySearch* method is not shown in the provided code, but it is expected to implement the binary search algorithm and return the index of the target product name if found or -1 if not found. After performing the binary search, the code checks the return value of the *binarySearch* method. If the index is not -1, it means the target product name is found in the *dataList*. The code gets the corresponding data from *dataList*, which contains the entire data record in the format "ID,name,price$". It then splits this record using the *split*() method to extract the product ID and price. Finally, it prints the search result to the console, displaying the product name, ID, and price.

The code also measures the runtime of the "Search" method (excluding user input) using a Timer class. It prints the elapsed time in nanoseconds. And also print the memory space used by the function using *getMemoryUsedDuringMethod*(). This code allows the user to search for a specific product by its name in the sorted data. If the product is found, it displays the product name, ID, price, and quantity. If the product is not found, it informs the user that the product name was not found. The code uses a binary search algorithm for efficient searching.



```
                    case 4:
                        System.out.println("Exiting the program.");
                        break;
                    default:
                        System.out.println("Invalid choice.");
                        break;
                }
            } while (choice != 4);

            scanner.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Inside this case 4 block, when the user selects option 4 from the menu, the code prints the message "Exiting the program." to inform the user that the program is terminating.

In this part of code also includes a default case in the switch statement to handle any invalid choices made by the user. If the user enters a choice other than 1, 2, 3, or 4, the default case will be executed. It prints the message "Invalid choice." to inform the user that the input is not recognized as a valid option. After processing the user's choice, the code checks if the choice variable is equal to 4. If it is, the loop will terminate, and the program will exit. Otherwise, the loop will continue to prompt the user for input until they choose to exit (select option 4). After the loop ends, the code closes the Scanner object (scanner) to release any resources associated with it.

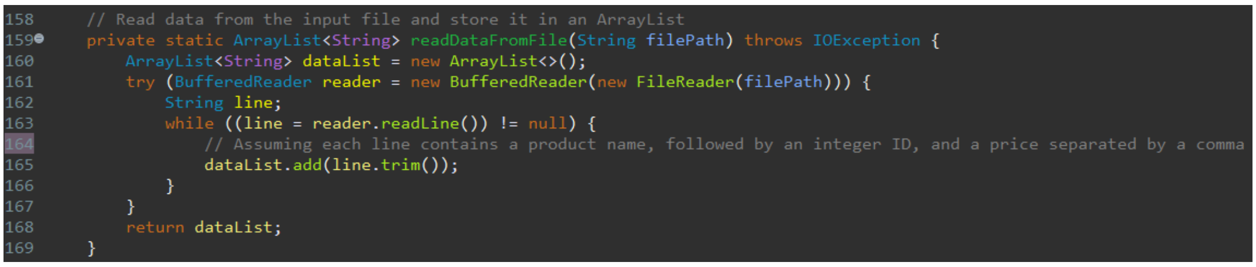
In this part of code, it is enclosed in a try-catch block, where it catches any *IOException* that may occur during file read/write operations or if any other I/O-related error occurs. If such an exception occurs, it prints the exception's stack trace to the console. This allows the program to handle potential errors gracefully instead of crashing abruptly.

<u>Function in Main Fucntion:</u>

1- **readDataFromFile() :**

This method takes a *filePath* as input, which is the path to the input file containing product data (daatbase). Each line contains a product name, followed by an integer product ID, a price, and a quantity all separated by commas. Inside this method, a new *ArrayList<String>* named *dataList* is created. This list will be used to store the product data read from the input file. The method will use a try-with-resources statement to create a *BufferedReader* named reader. The *BufferedReader* is used to efficiently read data from the input file.

Then it enters a while loop, where it reads each line from the input file using the *reader.readLine()* method. If there are lines to read (*line* is not null), the loop continues. For each line read from the file, it is trimmed to remove any leading or trailing commas using *line.trim()*. This is done to clean up the input data. The cleaned-up line (product data) is then added to the *dataList* using the add method of *ArrayList*. The loop continues until all lines in the input file have been read and added to the *dataList*. Once the reading is complete, the *dataList* containing the product data is returned. The *readDataFromFile* method reads product data from a given input file, where each line represents a product with its name, ID, and price. It stores this data in an *ArrayList<String>*, which can then be used for further processing or manipulation within the program.

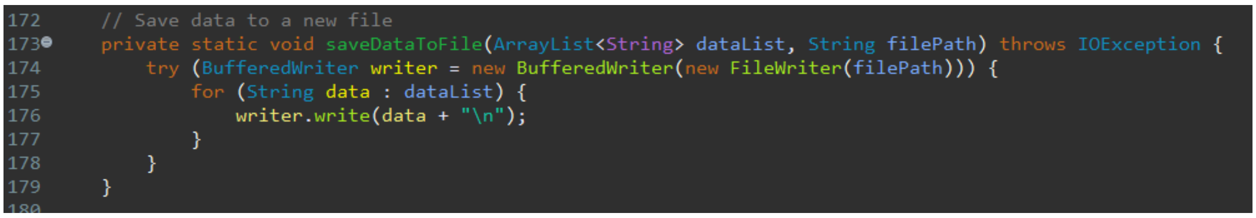


```java
    // Read data from the input file and store it in an ArrayList
    private static ArrayList<String> readDataFromFile(String filePath) throws IOException {
        ArrayList<String> dataList = new ArrayList<>();
        try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
            String line;
            while ((line = reader.readLine()) != null) {
                // Assuming each line contains a product name, followed by an integer ID, and a price separated by a comma
                dataList.add(line.trim());
            }
        }
        return dataList;
    }
```

2- saveDataToFile() :

This method takes two arguments: *dataList*, which is an ArrayList of strings containing the data that needs to be saved, and *filePath*, which is a string representing the path to the output file where the data will be saved. The method uses a try-with-resources block to create a BufferedWriter that will be used to write data to the specified file. The try-with-resources ensures that the BufferedWriter is closed properly after the execution of the block. It enters a for loop that iterates over each element (string) in the *dataList*.

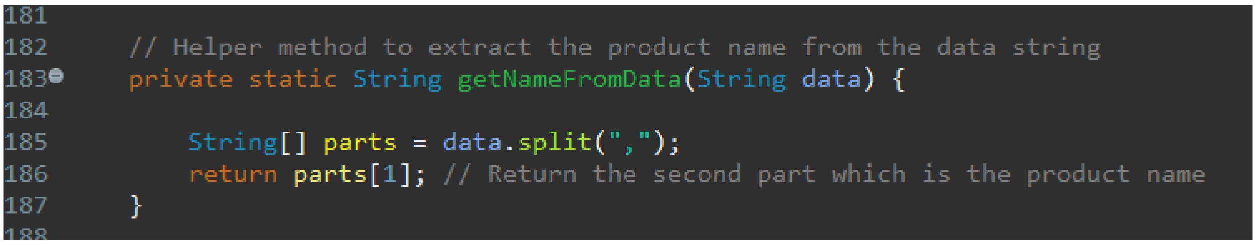
Inside the for loop, it writes each element (data) to the BufferedWriter by using *writer.write(data + "\n")*. The *"\n"* is added to insert a new line after each data element, so each data element will be written on a new line in the output file. When the loop finishes writing all the data elements to the file, the method closes the BufferedWriter, ensuring that the data is removed and saved to the output file. This *saveDataToFile* method takes an ArrayList of strings containing data and writes each element of the list to the output file, with each element written on separate line. If the file already exists, it will be overwritten with the new data. If the file doesn't exist, a new file will be created at the specified file path, and the data will be written to it. The method will throw an IOException if there are any issues with writing the data to the file, such as if the file is read-only or if there are permission-related problems.



```
    // Save data to a new file
    private static void saveDataToFile(ArrayList<String> dataList, String filePath) throws IOException {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(filePath))) {
            for (String data : dataList) {
                writer.write(data + "\n");
            }
        }
    }
```

3- **getNameFromData() :**

This method is a helper function designed to extract the product name from a given data string. This method takes a single parameter data, which is a string representing a single entry of product data. Each part of the data is separated by a space character. Inside the method, the *split(",")* method is used on the data string to split it into an array of substrings based on the space character. This will divide the data into separate parts, where index [0] will be the product ID, index [1] will be the product name, and index [2] will be the price. The method returns index [1], which is the second element of the parts array, representing the product name. This is because array indices start from 0, so index [0] contains the product ID, index [1] contains the product name, index [2] contains the price and index[3] contains the quantity. This method takes a data string in the format "*product_id product_name price_quantity*" and extracts the product name from it. It then returns the product name as a separate string, allowing it to be used for comparison or other purposes in the program.



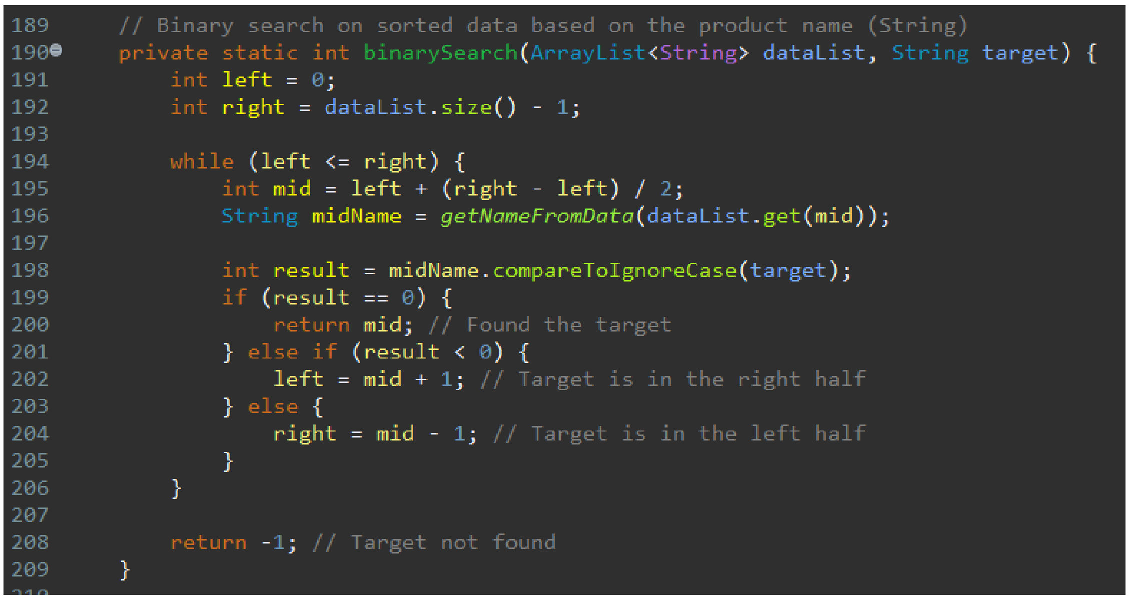
```

    // Helper method to extract the product name from the data string
    private static String getNameFromData(String data) {

        String[] parts = data.split(",");
        return parts[1]; // Return the second part which is the product name
    }
```

4- **binarySearch() :**

This method takes two parameters: *dataList: An ArrayList<String>* that contains the sorted product data based on the product name and *target*: The product name (String) that needs to be searched for in the *dataList*. There are two integer variables, *left* and *right*, are initialized to the start and end indices of the list, respectively. *left* is set to *0*, representing the first element of the list, and *right* is set to *dataList.size() - 1*, representing the last element of the list. This method enters a while loop, which continues until *left* is less than or equal to *right*. This while loop performs the binary search on the sorted list.

Inside the while loop, the variable *mid* is calculated as the middle index between *left* and *right*, using the formula *mid = left + (right - left) / 2*. This calculation ensures that the midpoint is not allowing integer overflow. The product name at the middle index *mid* is extracted from the *dataList* using the *getNameFromData* method and stored in the variable *midName*. The method then compares *midName* with the target product name using *compareToIgnoreCase*. The *compareToIgnoreCase* method compares two strings lexicographically, ignoring case differences. The result is stored in the variable result. If result is equal to *0*, it means that the target product name has been found at the middle index *mid*, and the method returns *mid*. If result is less than *0*, it means that the target product name is lexicographically greater than *midName*, so the target must be in the right half of the remaining list. The left index is updated to *mid + 1* to search in the right half. If result is greater than *0*, it means that the target product name is lexicographically less than *midName*, so the target must be in the left half of the remaining list. The right index is updated to *mid - 1* to search in the left half.

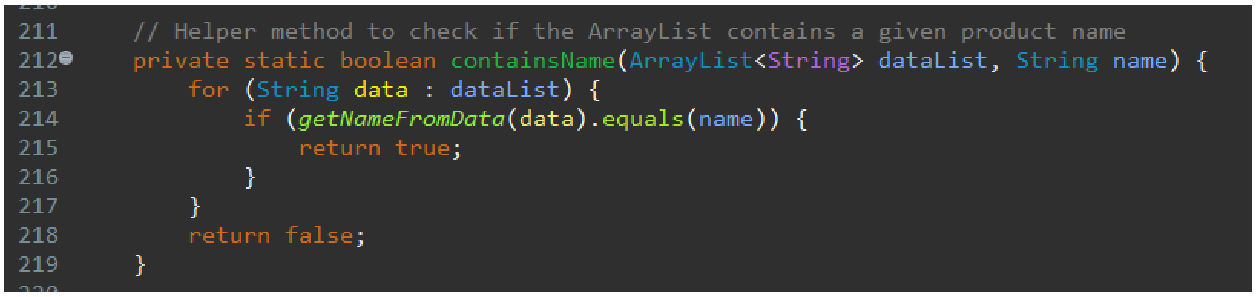
The while loop continues to execute the binary search, repeatedly narrowing down the search range by dividing the list in half, until the target product name is found, or the search range becomes empty (*left > right*). If the target product name is not found within the loop, the method returns *-1*, indicating that the target is not present in the *dataList*. This *binarySearch* method searches for a specific product name (target) in a sorted list of product data (*dataList*) using the binary search algorithm. It efficiently finds the index of the target product name if it exists in the list or returns -1 if the target is not present.



```java
    // Binary search on sorted data based on the product name (String)
    private static int binarySearch(ArrayList<String> dataList, String target) {
        int left = 0;
        int right = dataList.size() - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;
            String midName = getNameFromData(dataList.get(mid));

            int result = midName.compareToIgnoreCase(target);
            if (result == 0) {
                return mid; // Found the target
            } else if (result < 0) {
                left = mid + 1; // Target is in the right half
            } else {
                right = mid - 1; // Target is in the left half
            }
        }

        return -1; // Target not found
    }
```

5- containsName() :

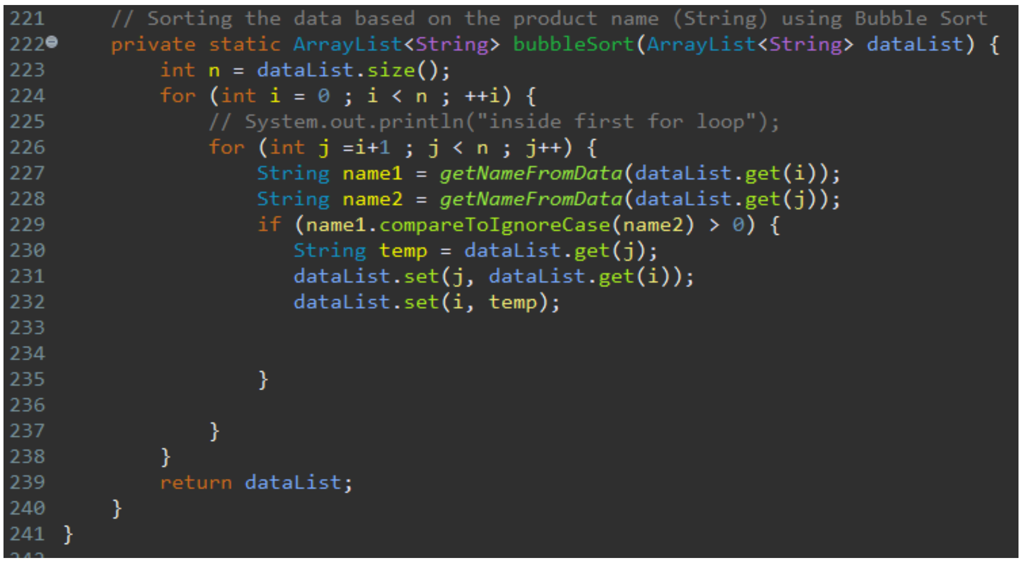
This method takes two parameters: *dataList*: *An ArrayList<String>* that contains the product data and *name*: The product name (String) that needs to be checked for existence in the *dataList*. The method uses a for-each loop to iterate through each element (data) in the *dataList*. For each data element, the method calls the *getNameFromData* method, passing the data as an argument. The *getNameFromData* method extracts the product name from the data string, which is then compared with the given name using the *equals* method. If the extracted product name (*getNameFromData(data)*) is *equal* to the given name, it means that the name exists in the *dataList*. In this case, the method immediately returns *true*, indicating that the product name is found in the list. If the for loop finishes without finding a matching product name, the method returns *false*, indicating that the name does not exist in the *dataList*. The *containsName* method checks if a given product name (name) exists in the *ArrayList<String> dataList*. It iterates through the list and compares each product name in the list with the given name. If a match is found, it returns *true*, and if no match is found, it returns *false*. This method is useful for checking if a particular product name is already present in the list before adding a new product or performing other operations.



```java
    // Helper method to check if the ArrayList contains a given product name
    private static boolean containsName(ArrayList<String> dataList, String name) {
        for (String data : dataList) {
            if (getNameFromData(data).equals(name)) {
                return true;
            }
        }
        return false;
    }
```

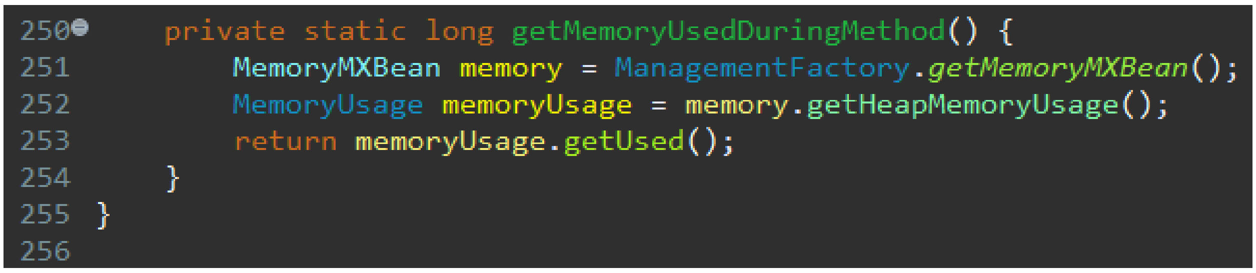
6- **bubbleSort():** :

The *bubbleSort*() method is an implementation of the Bubble Sort algorithm, which aims to sort data based on the product name (String) in ascending order. It takes an *ArrayList* of strings, *dataList*, as input, with each element representing a product entry in the format "ID,name,price$". The method calculates the size of the *dataList* and enters two nested for loops to compare adjacent product names and swap them if necessary. The outer for loop iterates from 0 to n-1, representing the current element being compared and placed in the correct position. The inner for loop runs from i+1 to n-1, representing the elements compared with the element at index i. Inside the inner loop, it extracts product names using the *getNameFromData*() method, and then compares them using *compareToIgnoreCase*(). If the first name is lexicographically greater than the second, it indicates they are in the wrong order, and the method swaps their positions. The process continues until all elements are compared, and the list is sorted based on product names in ascending order. The original *dataList* is modified during the sorting process. Finally, the sorted *dataList* is returned as the output of the method. This algorithm efficiently reorders the data according to the product names, making the search easier and retrieval of products in an ordered manner.



```java
    // Sorting the data based on the product name (String) using Bubble Sort
    private static ArrayList<String> bubbleSort(ArrayList<String> dataList) {
        int n = dataList.size();
        for (int i = 0 ; i < n ; ++i) {
            // System.out.println("inside first for loop");
            for (int j =i+1 ; j < n ; j++) {
                String name1 = getNameFromData(dataList.get(i));
                String name2 = getNameFromData(dataList.get(j));
                if (name1.compareToIgnoreCase(name2) > 0) {
                   String temp = dataList.get(j);
                   dataList.set(j, dataList.get(i));
                   dataList.set(i, temp);


                }

            }
        }
        return dataList;
    }
}
```

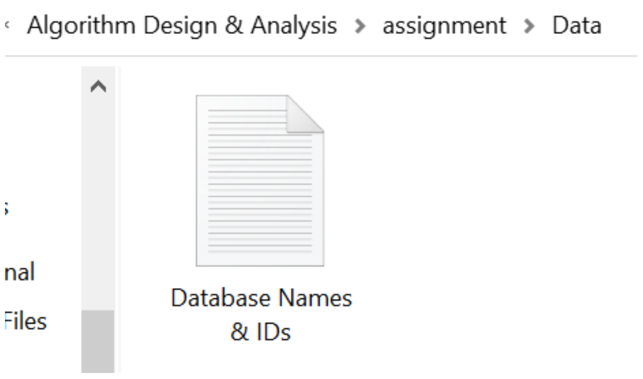
7- *getMemoryUsedDuringMethod*() : This method calculates the amount of memory used by accessing the *MemoryMXBean*, which provides memory usage information. Specifically, it retrieves the heap memory usage using *getHeapMemoryUsage*() and returns the number of bytes of heap memory currently in use with *memoryUsage.getUsed*(). This functionality allows us to monitor and analyze memory consumption during the execution of main methods, aiding in memory profiling and optimization. In conclusion, this method allows us to monitor and retrieve the amount of heap memory used at a specific point in our Java code. It's often useful for memory profiling and optimization to understand memory consumption during program execution.

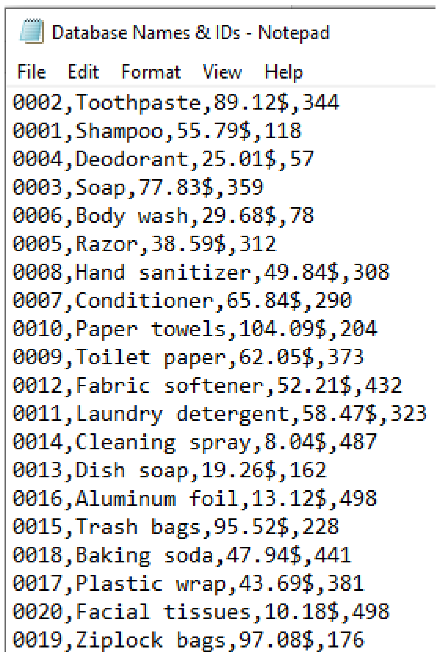


```java
    private static long getMemoryUsedDuringMethod() {
        MemoryMXBean memory = ManagementFactory.getMemoryMXBean();
        MemoryUsage memoryUsage = memory.getHeapMemoryUsage();
        return memoryUsage.getUsed();
    }
}

```

<u>Functionality of the code:</u>

At the beginning, we have a file that contains the database (input path) and the program will go to that path to get the database called "*Database Names & IDs*" (unsorted by Names and IDs)



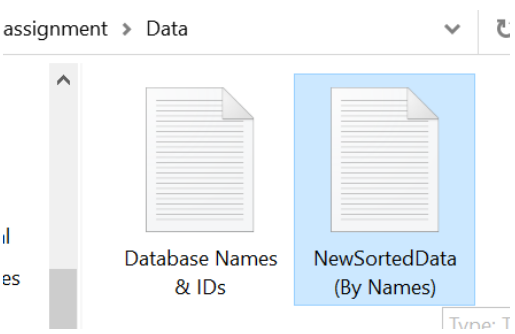


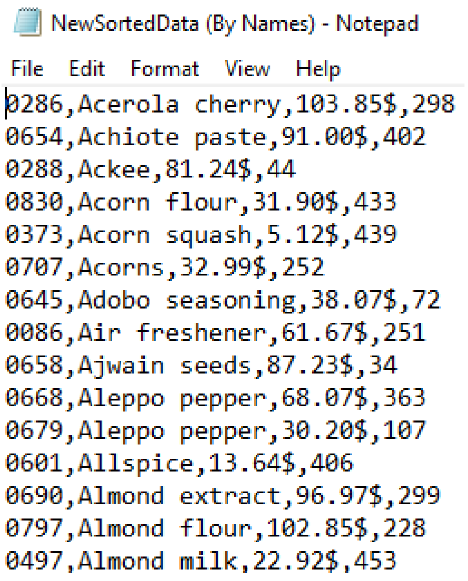
Database Names & IDs - Notepad

File Edit Format View Help

```
0002,Toothpaste,89.12$,344
0001,Shampoo,55.79$,118
0004,Deodorant,25.01$,57
0003,Soap,77.83$,359
0006,Body wash,29.68$,78
0005,Razor,38.59$,312
0008,Hand sanitizer,49.84$,308
0007,Conditioner,65.84$,290
0010,Paper towels,104.09$,204
0009,Toilet paper,62.05$,373
0012,Fabric softener,52.21$,432
0011,Laundry detergent,58.47$,323
0014,Cleaning spray,8.04$,487
0013,Dish soap,19.26$,162
0016,Aluminum foil,13.12$,498
0015,Trash bags,95.52$,228
0018,Baking soda,47.94$,441
0017,Plastic wrap,43.69$,381
0020,Facial tissues,10.18$,498
0019,Ziplock bags,97.08$,176
```

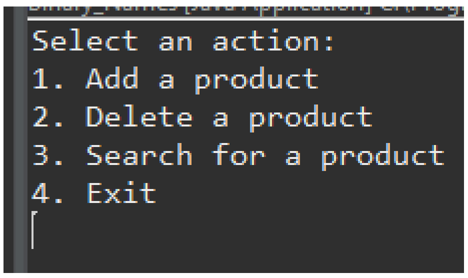
When we start to run the program. It automatically starts by creating a new .txt file (output file) called "*NewSortedData (By Names)*" that sorts all the data inside it according to their Name using the bubble sort we explained above.




NewSortedData (By Names) - Notepad
File Edit Format View Help

```
0286,Acerola cherry,103.85$,298
0654,Achiote paste,91.00$,402
0288,Ackee,81.24$,44
0830,Acorn flour,31.90$,433
0373,Acorn squash,5.12$,439
0707,Acorns,32.99$,252
0645,Adobo seasoning,38.07$,72
0086,Air freshener,61.67$,251
0658,Ajwain seeds,87.23$,34
0668,Aleppo pepper,68.07$,363
0679,Aleppo pepper,30.20$,107
0601,Allspice,13.64$,406
0690,Almond extract,96.97$,299
0797,Almond flour,102.85$,228
0497,Almond milk,22.92$,453
```

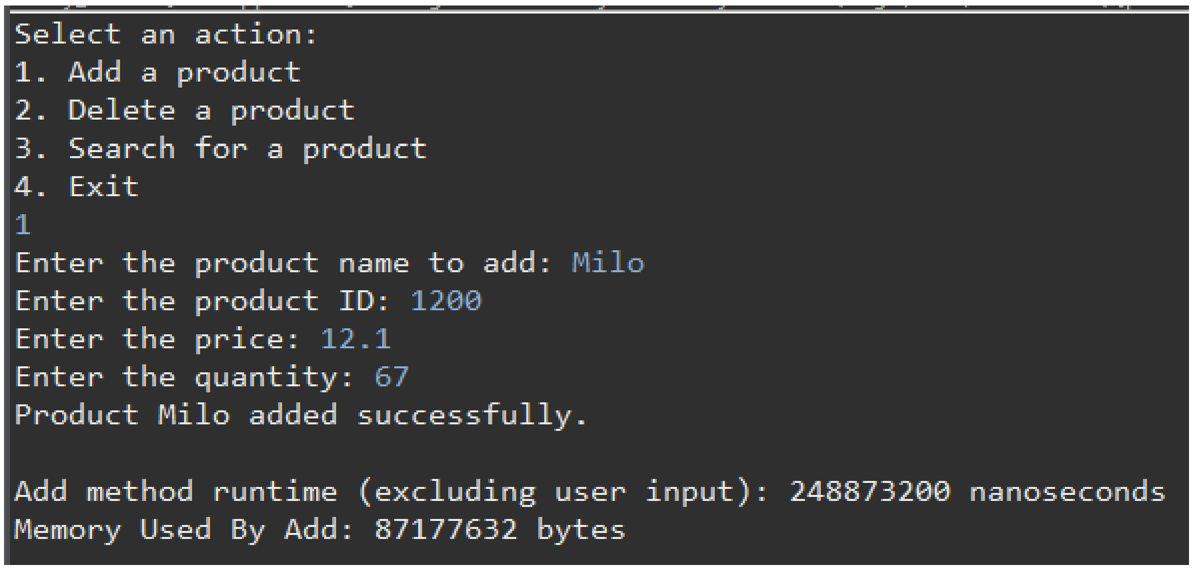
All the work starts in the console. This message is printed in the console, where the user must interact with the program to choose the command he wants:



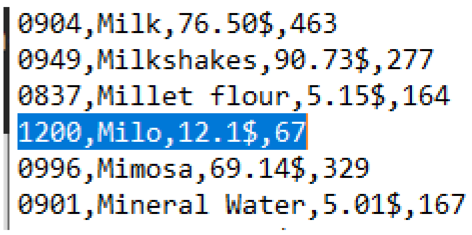
```
Select an action:
1. Add a product
2. Delete a product
3. Search for a product
4. Exit
```

We take the commands one by one. We start by the first one:

1- Add a product:

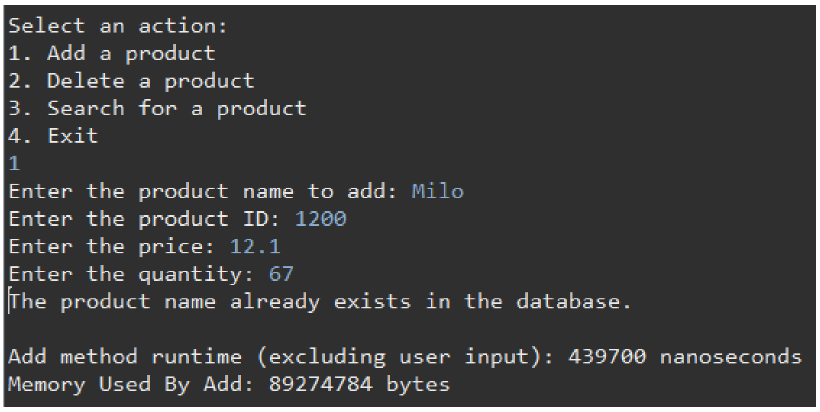


```
Select an action:
1. Add a product
2. Delete a product
3. Search for a product
4. Exit
1
Enter the product name to add: Milo
Enter the product ID: 1200
Enter the price: 12.1
Enter the quantity: 67
Product Milo added successfully.

Add method runtime (excluding user input): 248873200 nanoseconds
Memory Used By Add: 87177632 bytes
```



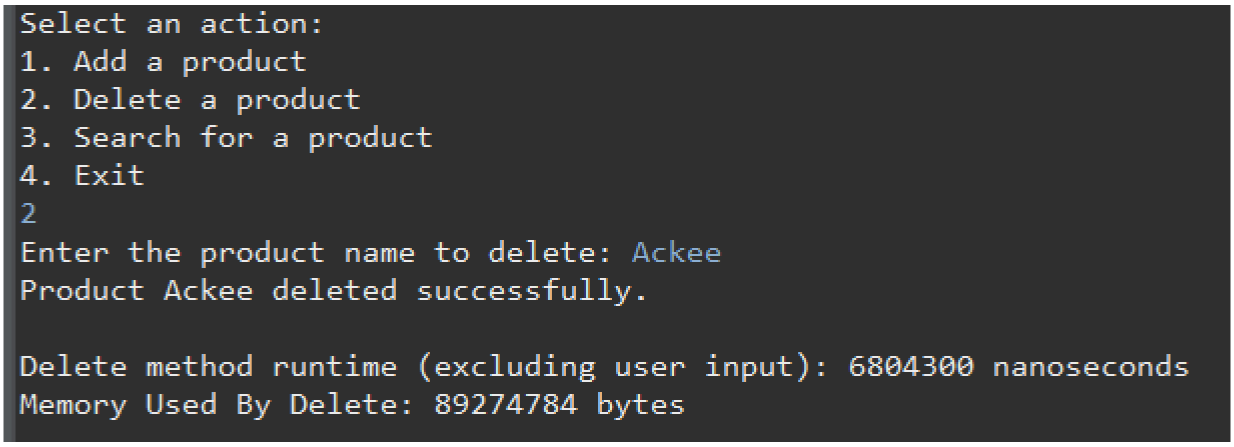
```
0904,Milk,76.50$,463
0949,Milkshakes,90.73$,277
0837,Millet flour,5.15$,164
1200,Milo,12.1$,67
0996,Mimosa,69.14$,329
0901,Mineral Water,5.01$,167
```

When the user chooses to add, the program asks him to enter the Names he wants to add. Then it asks him to write the ID number of the product. And finally, the price of the product. The program will first check if this Name already exists in the database or not. If it doesn't exist, it will show that the Product is added successfully and will print the runtime of the method in nanoseconds (excluding the user time) and the memory used by the method. And it will automatically update the output file "*NewSortedData (By Names)*" If the Name already exists, the program will print that the Product name already exists in the database. And it will also print the runtime and the memory used.

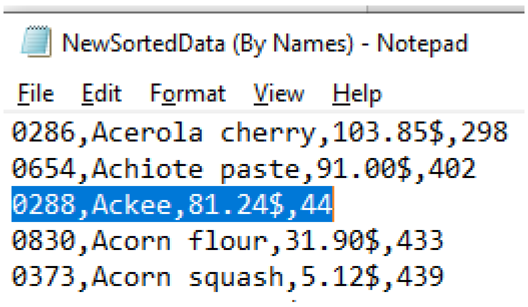


```
Select an action:
1. Add a product
2. Delete a product
3. Search for a product
4. Exit
1
Enter the product name to add: Milo
Enter the product ID: 1200
Enter the price: 12.1
Enter the quantity: 67
The product name already exists in the database.

Add method runtime (excluding user input): 439700 nanoseconds
Memory Used By Add: 89274784 bytes
```

2- Delete a product:

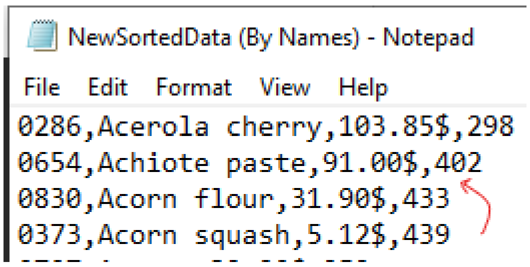


```
Select an action:
1. Add a product
2. Delete a product
3. Search for a product
4. Exit
2
Enter the product name to delete: Ackee
Product Ackee deleted successfully.

Delete method runtime (excluding user input): 6804300 nanoseconds
Memory Used By Delete: 89274784 bytes
```



NewSortedData (By Names) - Notepad
File Edit Format View Help

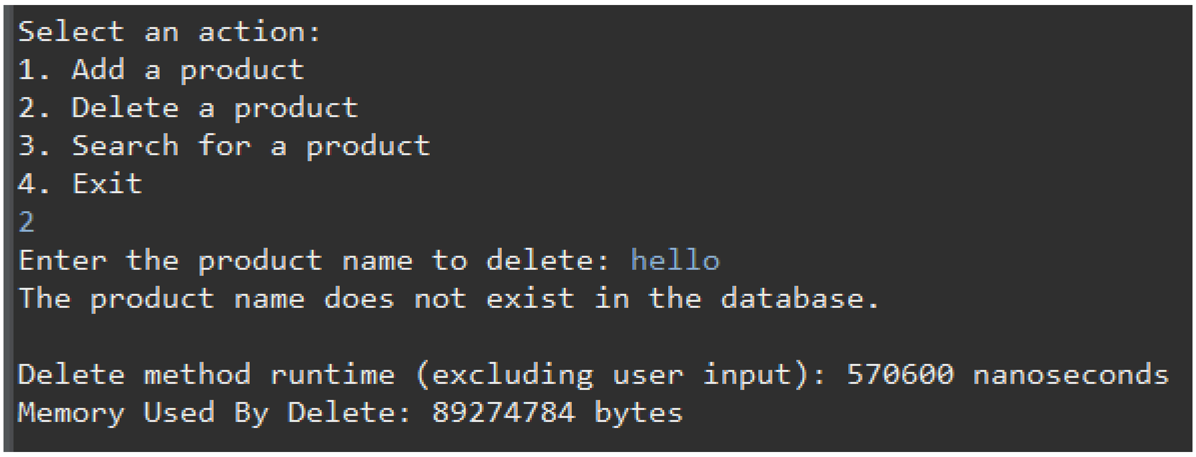
```
0286,Acerola cherry,103.85$,298
0654,Achiote paste,91.00$,402
0288,Ackee,81.24$,44
0830,Acorn flour,31.90$,433
0373,Acorn squash,5.12$,439
```



NewSortedData (By Names) - Notepad
File Edit Format View Help

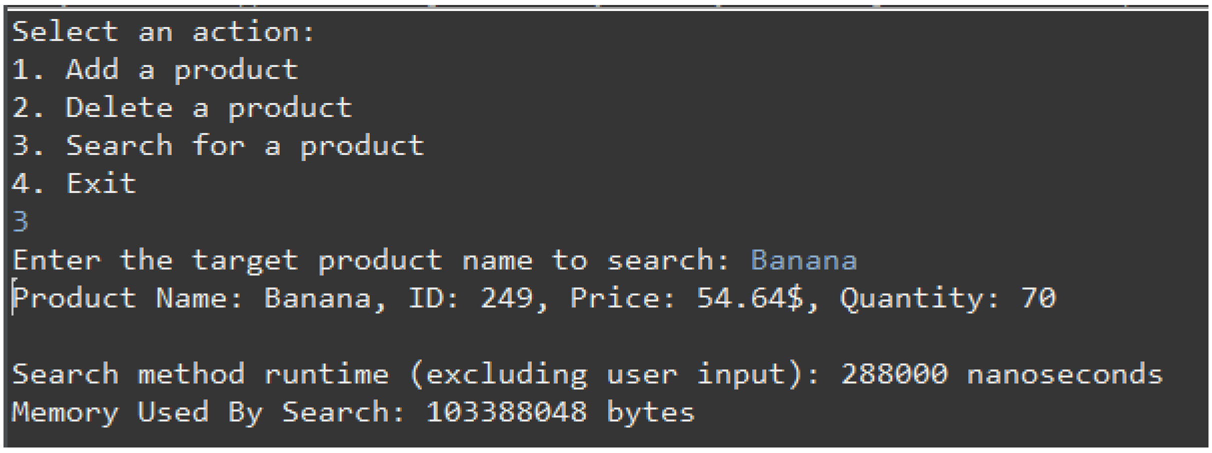
```
0286,Acerola cherry,103.85$,298
0654,Achiote paste,91.00$,402
0830,Acorn flour,31.90$,433
0373,Acorn squash,5.12$,439
```

When the user chooses to delete, the program asks him to enter the Name he wants to delete. The program will first check if this Name already exists in the database or not. If the Name exists in the database, it will perform the delete action and remove it from the database and will print a successful message and the runtime of the method in nanoseconds (excluding the user time) and the memory used by the method. And it will automatically update the output file "*NewSortedData (By Names)*". If it doesn't exist, it will show that the Name does not exist in the database. And it will also print the runtime and the memory used.



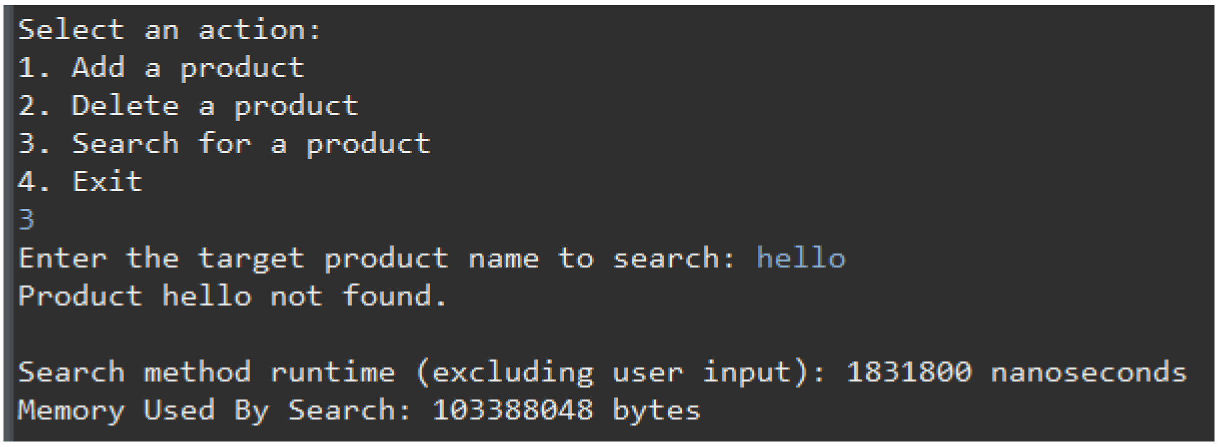
```
Select an action:
1. Add a product
2. Delete a product
3. Search for a product
4. Exit
2
Enter the product name to delete: hello
The product name does not exist in the database.

Delete method runtime (excluding user input): 570600 nanoseconds
Memory Used By Delete: 89274784 bytes
```

3- Search for a product:



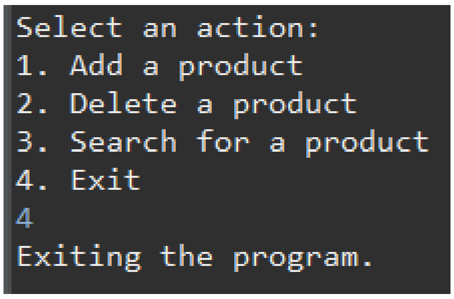
```
Select an action:
1. Add a product
2. Delete a product
3. Search for a product
4. Exit
3
Enter the target product name to search: Banana
Product Name: Banana, ID: 249, Price: 54.64$, Quantity: 70

Search method runtime (excluding user input): 288000 nanoseconds
Memory Used By Search: 103388048 bytes
```

When the user chooses to search, the program asks him to enter the Name he wants to search for. The program will first check if this Name already exists in the database or not. If the Name exists in the database, it will print all the data related to this Name from the database (ID, Name, Price, Quantity) and will print the runtime of the method in nanoseconds (excluding the user time) and the memory used by the method. And it will automatically update the output file "*NewSortedData (By Names)*". If it doesn't exist, it will show that the Product is not found in the database. And it will also print the runtime and the memory used.



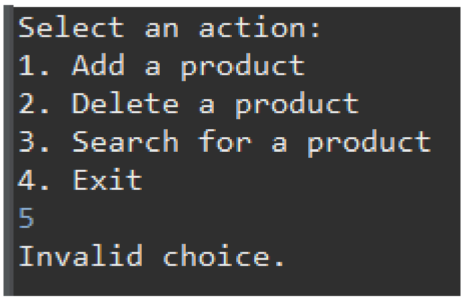
```
Select an action:
1. Add a product
2. Delete a product
3. Search for a product
4. Exit
3
Enter the target product name to search: hello
Product hello not found.

Search method runtime (excluding user input): 1831800 nanoseconds
Memory Used By Search: 103388048 bytes
```

4- Exit:



```
Select an action:
1. Add a product
2. Delete a product
3. Search for a product
4. Exit
4
Exiting the program.
```

When the user chooses to exit. The program terminates everything and stops running. and it leaves the output file. If you enter any other option that is not in the option, you get a message saying: Invalid choice. And get you back to the options of action.

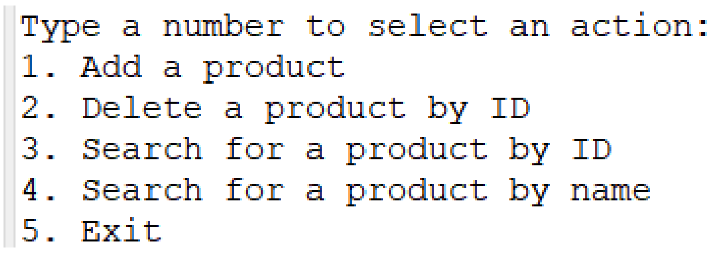


```
Select an action:
1. Add a product
2. Delete a product
3. Search for a product
4. Exit
5
Invalid choice.
```

4.1.2 APPLICATION 2: LINKED LIST + LINEAR SEARCH + MERGE SORT

How the program works:

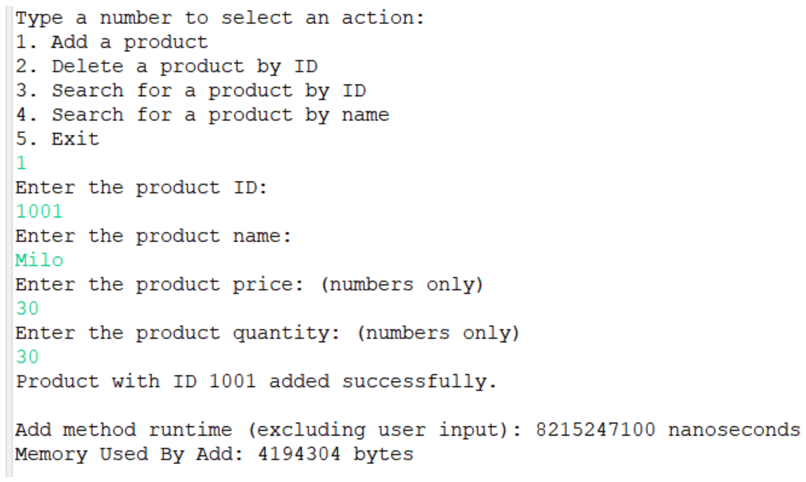
Menu



```
Type a number to select an action:
1. Add a product
2. Delete a product by ID
3. Search for a product by ID
4. Search for a product by name
5. Exit
```

The figure above shows when the program is started, the program will prompt the user to choose from 5 actions, which includes adding a product, deleting a product by ID, searching for a product by ID, searching for a product by name, and exit.

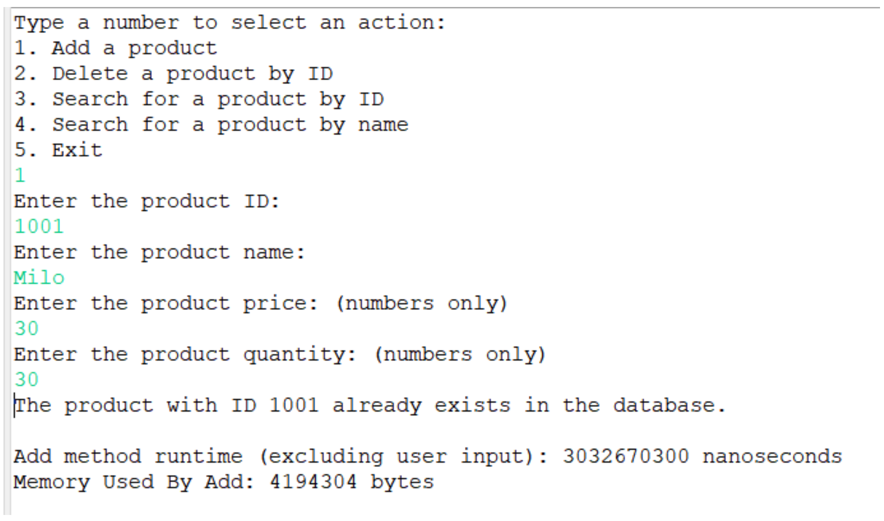
Add a product (successful scenario)



```
Type a number to select an action:
1. Add a product
2. Delete a product by ID
3. Search for a product by ID
4. Search for a product by name
5. Exit
1
Enter the product ID:
1001
Enter the product name:
Milo
Enter the product price: (numbers only)
30
Enter the product quantity: (numbers only)
30
Product with ID 1001 added successfully.

Add method runtime (excluding user input): 8215247100 nanoseconds
Memory Used By Add: 4194304 bytes
```

To choose, the user will have to key in the number that matches with the corresponding action. In the figure above, the user typed 1, and this choice is to add a product into the database. The program will ask the user to enter the product ID, name, price and quantity. If the product ID is not found in the database, the program will add the product ID together with its details into the database.

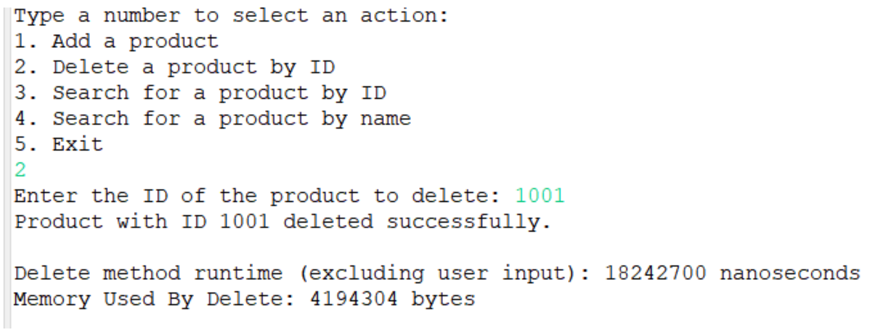
Add a product (unsuccessful scenario)



```
Type a number to select an action:
1. Add a product
2. Delete a product by ID
3. Search for a product by ID
4. Search for a product by name
5. Exit
1
Enter the product ID:
1001
Enter the product name:
Milo
Enter the product price: (numbers only)
30
Enter the product quantity: (numbers only)
30
The product with ID 1001 already exists in the database.

Add method runtime (excluding user input): 3032670300 nanoseconds
Memory Used By Add: 4194304 bytes
```

In the figure above, the user typed the same product ID, name, price and quantity again. This time, the product ID already exists in the database, so the program will not add the product ID together with its details into the database.

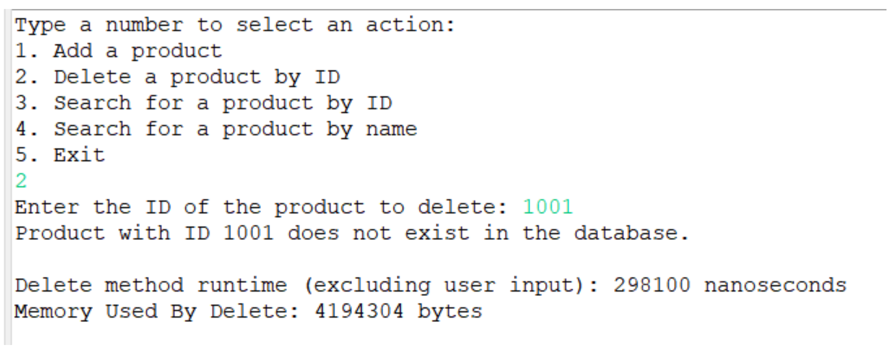
Delete a product (successful scenario)



```
Type a number to select an action:
1. Add a product
2. Delete a product by ID
3. Search for a product by ID
4. Search for a product by name
5. Exit
2
Enter the ID of the product to delete: 1001
Product with ID 1001 deleted successfully.

Delete method runtime (excluding user input): 18242700 nanoseconds
Memory Used By Delete: 4194304 bytes
```

In the figure above, the user typed 2, and this choice is to delete a product from the database. The program will ask the user to enter the product ID, and if the product ID exists in the database, the product will be deleted from the database.

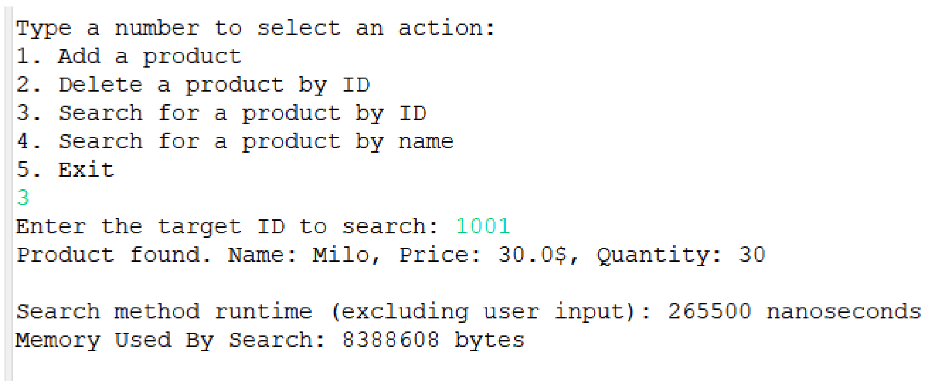
Delete a product (unsuccessful scenario)



```
Type a number to select an action:
1. Add a product
2. Delete a product by ID
3. Search for a product by ID
4. Search for a product by name
5. Exit
2
Enter the ID of the product to delete: 1001
Product with ID 1001 does not exist in the database.

Delete method runtime (excluding user input): 298100 nanoseconds
Memory Used By Delete: 4194304 bytes
```

In the figure above, the user typed the same product ID. This time, the product ID does not exist in the database, so the program will not delete the product from the database.

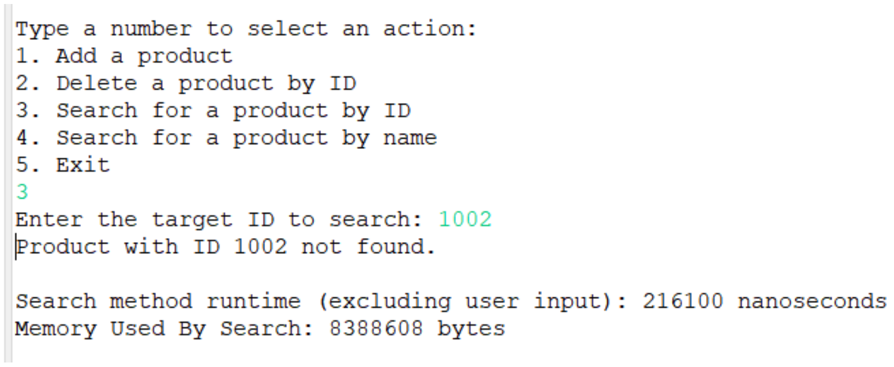
Search for a product by ID (successful scenario)



```
Type a number to select an action:
1. Add a product
2. Delete a product by ID
3. Search for a product by ID
4. Search for a product by name
5. Exit
3
Enter the target ID to search: 1001
Product found. Name: Milo, Price: 30.0$, Quantity: 30

Search method runtime (excluding user input): 265500 nanoseconds
Memory Used By Search: 8388608 bytes
```

In the figure above, the user typed 3, and this choice is to search for a product by ID. The program will ask the user to enter the product ID, and if the product ID exists in the database, the product name, price and quantity will be shown.

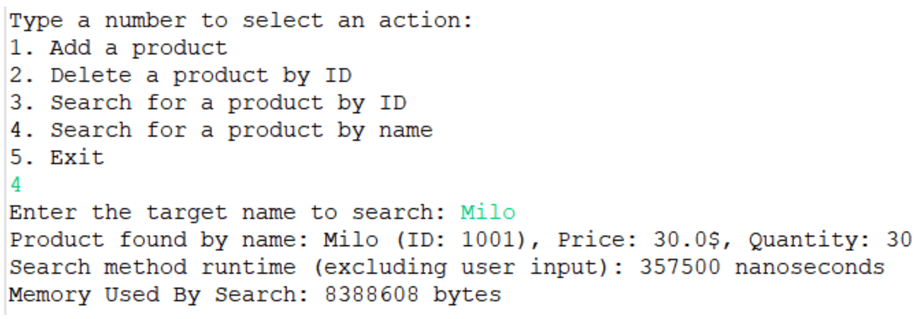
Search for a product by ID (unsuccessful scenario)



```
Type a number to select an action:
1. Add a product
2. Delete a product by ID
3. Search for a product by ID
4. Search for a product by name
5. Exit
3
Enter the target ID to search: 1002
Product with ID 1002 not found.

Search method runtime (excluding user input): 216100 nanoseconds
Memory Used By Search: 8388608 bytes
```

In the figure above, the user typed the product ID which does not exist in the database, so the program will not show any details of the product which does not exist.

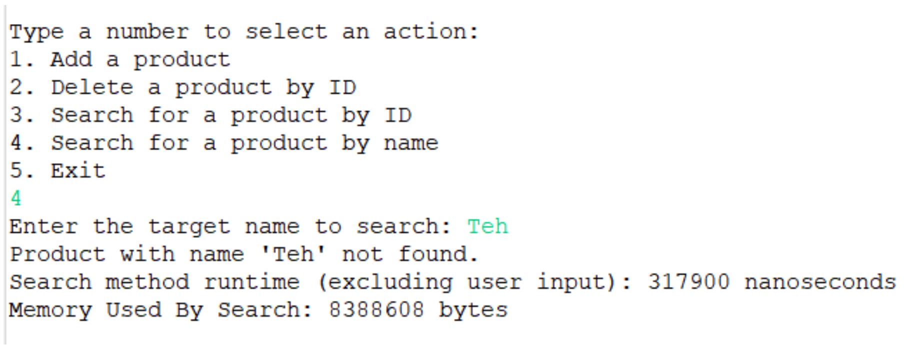
Search for a product by name (successful scenario)



```
Type a number to select an action:
1. Add a product
2. Delete a product by ID
3. Search for a product by ID
4. Search for a product by name
5. Exit
4
Enter the target name to search: Milo
Product found by name: Milo (ID: 1001), Price: 30.0$, Quantity: 30
Search method runtime (excluding user input): 357500 nanoseconds
Memory Used By Search: 8388608 bytes
```

In the figure above, the user typed 4, and this choice is to search for a product by name. The program will ask the user to enter the product name, and if the product name exists in the database, the product ID, price and quantity will be shown.

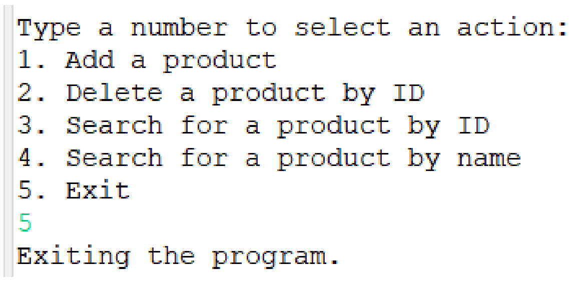
<u>Search for a product by name (unsuccessful scenario)</u>



```
Type a number to select an action:
1. Add a product
2. Delete a product by ID
3. Search for a product by ID
4. Search for a product by name
5. Exit
4
Enter the target name to search: Teh
Product with name 'Teh' not found.
Search method runtime (excluding user input): 317900 nanoseconds
Memory Used By Search: 8388608 bytes
```

In the figure above, the user typed the product name which does not exist in the database, so the program will not show any details of the product which does not exist.

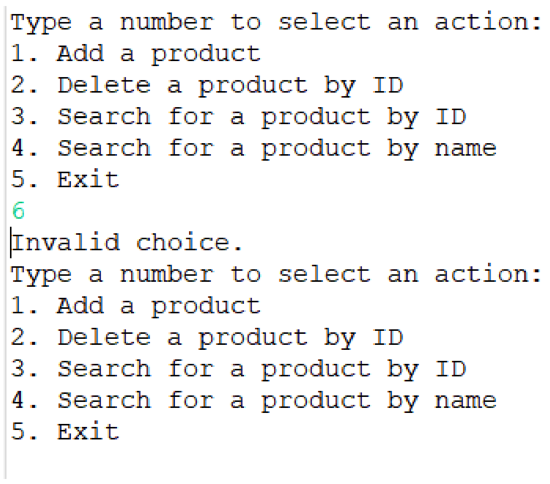
<u>Exit</u>



```
Type a number to select an action:
1. Add a product
2. Delete a product by ID
3. Search for a product by ID
4. Search for a product by name
5. Exit
5
Exiting the program.
```

In the figure above, the user typed 5, and this choice is to exit the program. The program will close the scanner and exit the program.

<u>Invalid choice</u>

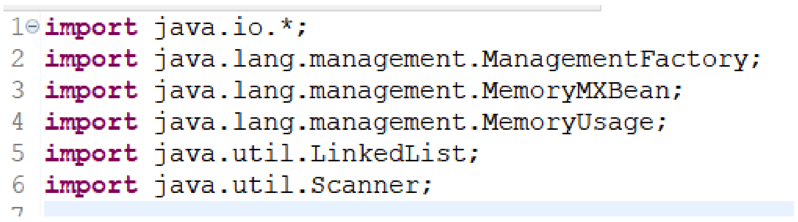


```
Type a number to select an action:
1. Add a product
2. Delete a product by ID
3. Search for a product by ID
4. Search for a product by name
5. Exit
6
Invalid choice.
Type a number to select an action:
1. Add a product
2. Delete a product by ID
3. Search for a product by ID
4. Search for a product by name
5. Exit
```

In the figure above, the user typed 6, and there are no functions implemented for number 6. In this case, the program will prompt the user to choose again. If the user typed anything other than numbers 1-5, the program will also prompt the user to choose again because there are no functions implemented for numbers other than 1-5.

CODING PART

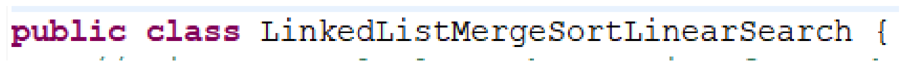
Import



```java
import java.io.*;
import java.lang.management.ManagementFactory;
import java.lang.management.MemoryMXBean;
import java.lang.management.MemoryUsage;
import java.util.LinkedList;
import java.util.Scanner;
```

These are the import statements, which allow the code to access classes and packages from the Java Standard Library. For my code, I have used java.io* for input/output operations, java.lang.management.* for memory management, java.util.LinkedList to access linked list and java.util.Scanner to use scanner class.

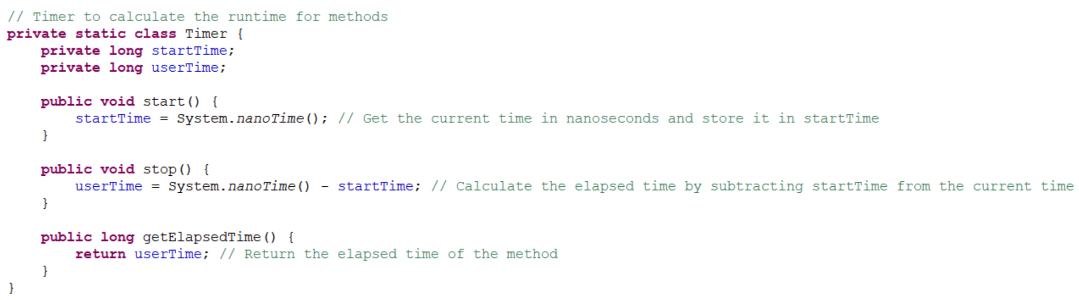
LinkedListMergeSortLinearSearch Class



```java
public class LinkedListMergeSortLinearSearch {
```

This is the beginning of the class named LinkedListMergeSortLinearSearch, inside the curly brackets {} will contain the main logic of the program.

Timer class



```java
// Timer to calculate the runtime for methods
private static class Timer {
    private long startTime;
    private long userTime;

    public void start() {
        startTime = System.nanoTime(); // Get the current time in nanoseconds and store it in startTime
    }

    public void stop() {
        userTime = System.nanoTime() - startTime; // Calculate the elapsed time by subtracting startTime from the current time
    }

    public long getElapsedTime() {
        return userTime; // Return the elapsed time of the method
    }
}
```

This Timer class is used to measure the runtime of methods. There are three methods, which are start() to start the timer, stop() to stop the timer and calculate elapsed time, and getElapsedTime() to get the elapsed time in nanoseconds.

Main method

```
public static void main(String[] args) {
    // Database file path
    String inputFilePath = "C:\\Users\\Acer\\Documents\\Y2S2\\Database.txt";
```

This is the main method, where the execution of the program starts. A String variable named inputFilePath is declared and it is initialized with the file path of the database text file.

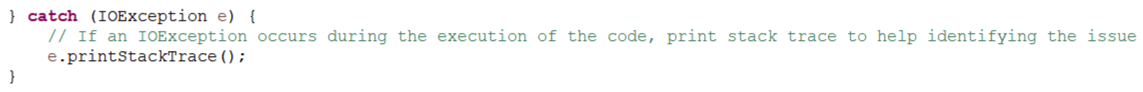
Try-catch block

```
try {
    // Read data from database and store it in LinkedList
    LinkedList<Product> dataList = readDataFromFile(inputFilePath);

    // Sorting the data using Merge Sort
    mergeSort(dataList);

    // Saving the sorted data to a new file
    saveDataToFile(dataList, inputFilePath);

    // Prompt user to choose an action
    Scanner scanner = new Scanner(System.in);
    int choice;
```

In this part of my code, I have used a try-catch block, because the code inside it may throw exceptions that need to be caught and handled.

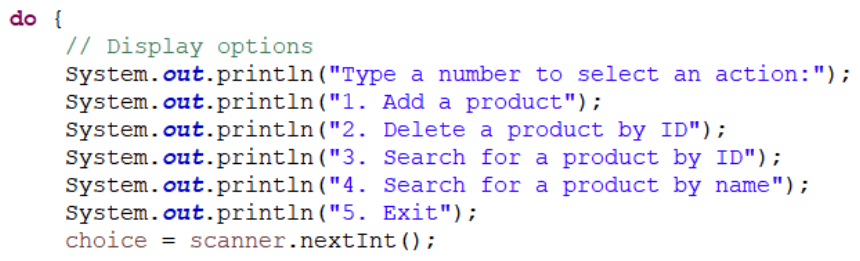
A LinkedList of Product objects named dataList is declared and initialized. It will be used to store the data read from the database text file. The readDataFromFile method is used to read data from the database text file and store it in the dataList linked list. The mergeSort method is used to sort the data in the dataList using the Merge Sort algorithm. The saveDataToFile method is used to save the sorted data from the dataList back to the inputFilePath, which is the database text file. The Scanner class named scanner is created and used to read input from the user and an int variable named choice is declared to store the user's selected action.

The methods mentioned above will be further explained in the methods part later in the report.

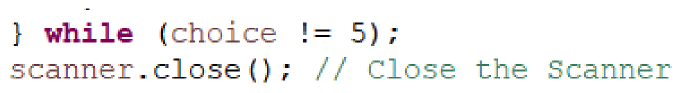

```
} catch (IOException e) {
    // If an IOException occurs during the execution of the code, print stack trace to help identifying the issue
    e.printStackTrace();
}
```

This catch block is to catch any IOException that might occur during execution. The stack trace will be printed to aid in debugging.

<u>Do-while loop</u>

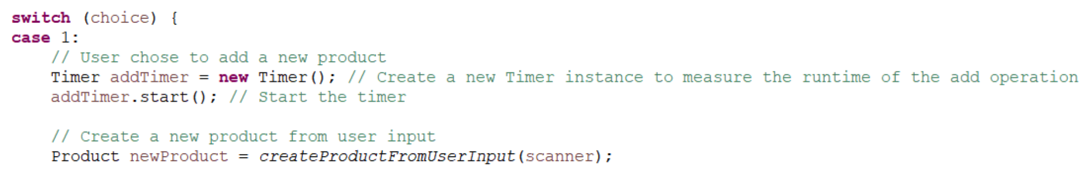

```
do {
    // Display options
    System.out.println("Type a number to select an action:");
    System.out.println("1. Add a product");
    System.out.println("2. Delete a product by ID");
    System.out.println("3. Search for a product by ID");
    System.out.println("4. Search for a product by name");
    System.out.println("5. Exit");
    choice = scanner.nextInt();
```

I have also used a do-while loop for my code. In the do part, the available options are displayed to the user using System.out.println(), and the user is prompted to enter a number to select an action. The user's choice is read using scanner.nextInt() and stored in the choice variable declared earlier.


```
} while (choice != 5);
scanner.close(); // Close the Scanner
```

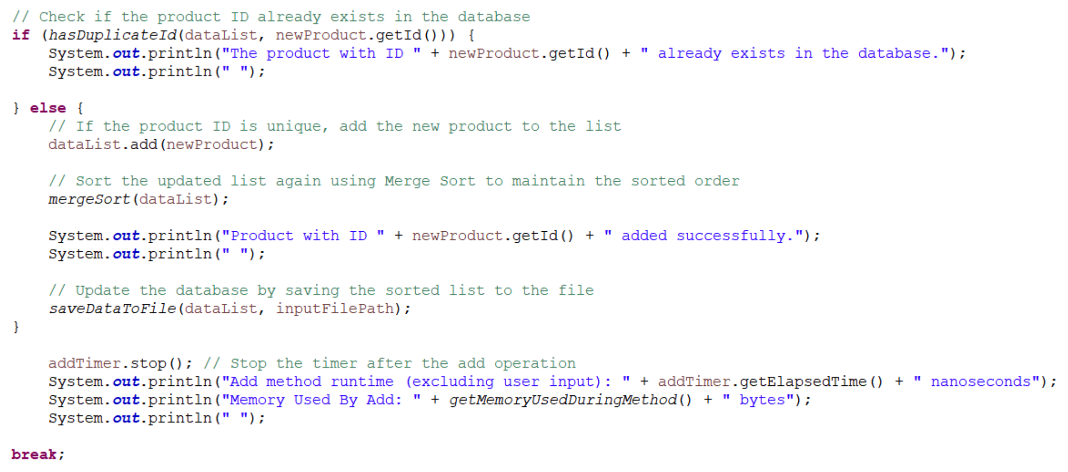
This while part keeps the loop running as long as the user does not choose option 5 to exit. If the user chooses 5, the Scanner object will be closed.

Case 1 (Adding a product)



```
switch (choice) {
case 1:
    // User chose to add a new product
    Timer addTimer = new Timer(); // Create a new Timer instance to measure the runtime of the add operation
    addTimer.start(); // Start the timer

    // Create a new product from user input
    Product newProduct = createProductFromUserInput(scanner);
```

A switch statement is used to run different blocks of code based on what the user has typed. If the user typed 1, then the user chose to add a new product. A Timer object named addTimer is created to measure the runtime of the addition operation. The createProductFromUserInput method is used to get the details of the new product from the user. This includes the product ID, name, price and quantity. This method will be explained later in the methods part. The product details are then stored in the newProduct variable.



```
    // Check if the product ID already exists in the database
    if (hasDuplicateId(dataList, newProduct.getId())) {
        System.out.println("The product with ID " + newProduct.getId() + " already exists in the database.");
        System.out.println(" ");

    } else {
        // If the product ID is unique, add the new product to the list
        dataList.add(newProduct);

        // Sort the updated list again using Merge Sort to maintain the sorted order
        mergeSort(dataList);

        System.out.println("Product with ID " + newProduct.getId() + " added successfully.");
        System.out.println(" ");

        // Update the database by saving the sorted list to the file
        saveDataToFile(dataList, inputFilePath);
    }

        addTimer.stop(); // Stop the timer after the add operation
        System.out.println("Add method runtime (excluding user input): " + addTimer.getElapsedTime() + " nanoseconds");
        System.out.println("Memory Used By Add:  " + getMemoryUsedDuringMethod() + " bytes");
        System.out.println(" ");

    break;
```

The hasDuplicateId method is used to check whether the product ID entered by the user already exists in the database or not. This method will be further explained later in the methods part. If the ID already exists, a message is printed informing the user that the product with that ID already exists. If the ID does not exist, the new product is added to the dataList, and the list is sorted again using the Merge Sort algorithm. This algorithm will be further explained later in the methods part. The sorted data is then saved back to the database file using the saveDataToFile method. This method will be further explained later in the methods part.

Then, the addTimer is stopped, and the runtime and memory used during the addition operation are displayed to the user.

Case 2 (Deleting a product)

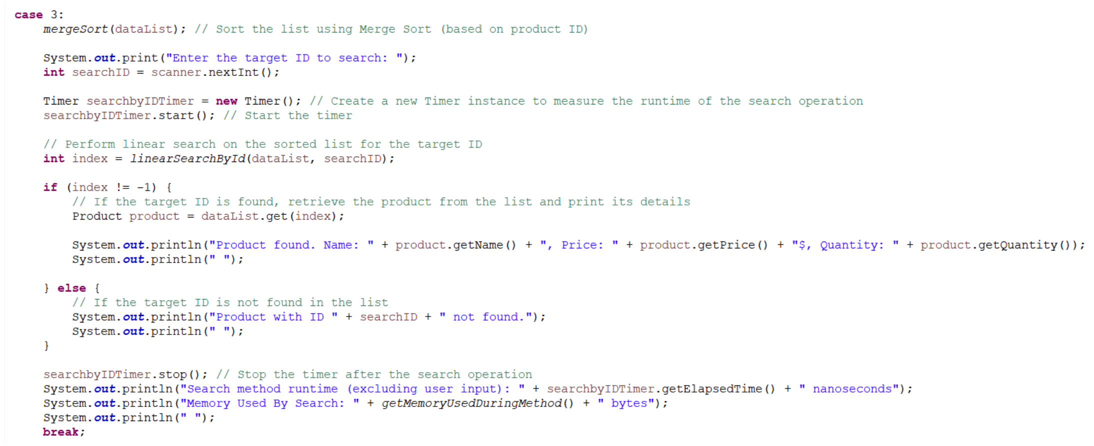
```java
case 2:
    // User chose to delete a product by ID
    System.out.print("Enter the ID of the product to delete: ");
    int deleteID = scanner.nextInt();

    Timer deleteTimer = new Timer(); // Create a new Timer instance to measure the runtime of the delete operation
    deleteTimer.start(); // Start the timer

    Product productToDelete = null;
    // Find the product with the specified ID in the list
    for (Product product : dataList) {
        if (product.getId() == deleteID) {
            productToDelete = product;
            break;
        }
    }

    // Check if the product with the specified ID exists in the list
    if (productToDelete != null) {
        // If the product exists, remove it from the list
        dataList.remove(productToDelete);

        System.out.println("Product with ID " + deleteID + " deleted successfully.");
        System.out.println(" ");

        // Update the output file by saving the updated list
        saveDataToFile(dataList, inputFilePath);

    } else {
        // If the product with the specified ID does not exist in the list
        System.out.println("Product with ID " + deleteID + " does not exist in the database.");
        System.out.println(" ");
    }

    deleteTimer.stop(); // Stop the timer after the delete operation
    System.out.println("Delete method runtime (excluding user input): " + deleteTimer.getElapsedTime() + " nanoseconds");
    System.out.println("Memory Used By Delete: " + getMemoryUsedDuringMethod() + " bytes");
    System.out.println(" ");
    break;
```

Case 2 represents the option to delete a product from the database by its ID. The user is asked to enter the ID of the product to delete. A Timer object deleteTimer is created to measure the runtime of the delete operation.

The program then searches for the product in the dataList that matches the specified deleteID. If the product is found (productToDelete is not null), then the product together with its details will be removed from the dataList. Then, the program will update the database with the updated dataList using the saveDataToFile method. If the product with the ID entered by the user is not found in the dataList, a message indicating the product does not exist is showed.

Then, the deleteTimer is stopped, and the runtime and memory used during the deletion operation are displayed to the user.

Case 3 (Searching for a product by ID)



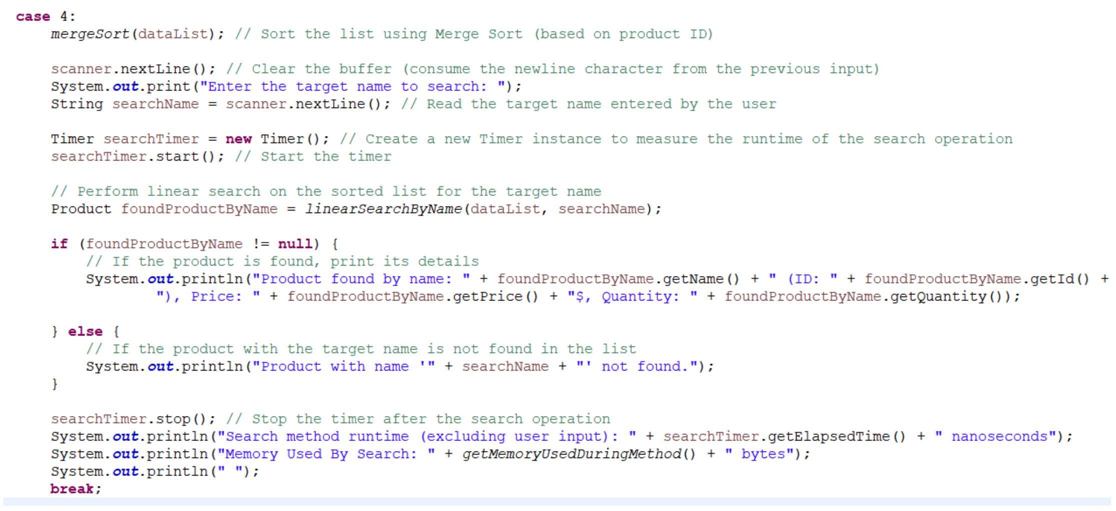
```java
case 3:
    mergeSort(dataList); // Sort the list using Merge Sort (based on product ID)

    System.out.print("Enter the target ID to search: ");
    int searchID = scanner.nextInt();

    Timer searchbyIDTimer = new Timer(); // Create a new Timer instance to measure the runtime of the search operation
    searchbyIDTimer.start(); // Start the timer

    // Perform linear search on the sorted list for the target ID
    int index = linearSearchById(dataList, searchID);

    if (index != -1) {
        // If the target ID is found, retrieve the product from the list and print its details
        Product product = dataList.get(index);

        System.out.println("Product found. Name: " + product.getName() + ", Price: " + product.getPrice() + "$, Quantity: " + product.getQuantity());
        System.out.println(" ");

    } else {
        // If the target ID is not found in the list
        System.out.println("Product with ID " + searchID + " not found.");
        System.out.println(" ");
    }

    searchbyIDTimer.stop(); // Stop the timer after the search operation
    System.out.println("Search method runtime (excluding user input): " + searchbyIDTimer.getElapsedTime() + " nanoseconds");
    System.out.println("Memory Used By Search: " + getMemoryUsedDuringMethod() + " bytes");
    System.out.println(" ");
    break;
```

Case 3 represents the option to search for a product in the database by its ID. Before searching, the dataList is sorted using the mergeSort method. Then, the user is prompted to enter the product ID to be searched. A Timer object searchbyIDTimer is created to measure the runtime of the search operation.

The program then performs a linear search on the sorted dataList to find the target product using the linearSearchById method. If the product with the specified ID is found (index != -1), the product details are printed. This includes the product name, price and quantity. If the product with the specified ID is not found, a message indicating the product is not found is displayed.

Then, the searchbyIDTimer is stopped, and the runtime and memory used during the searching operation are displayed to the user.

Case 4 (Searching a product by name)



```java
case 4:
    mergeSort(dataList); // Sort the list using Merge Sort (based on product ID)

    scanner.nextLine(); // Clear the buffer (consume the newline character from the previous input)
    System.out.print("Enter the target name to search: ");
    String searchName = scanner.nextLine(); // Read the target name entered by the user

    Timer searchTimer = new Timer(); // Create a new Timer instance to measure the runtime of the search operation
    searchTimer.start(); // Start the timer

    // Perform linear search on the sorted list for the target name
    Product foundProductByName = linearSearchByName(dataList, searchName);

    if (foundProductByName != null) {
        // If the product is found, print its details
        System.out.println("Product found by name: " + foundProductByName.getName() + " (ID: " + foundProductByName.getId() +
                "), Price: " + foundProductByName.getPrice() + "$, Quantity: " + foundProductByName.getQuantity());

    } else {
        // If the product with the target name is not found in the list
        System.out.println("Product with name '" + searchName + "' not found.");
    }

    searchTimer.stop(); // Stop the timer after the search operation
    System.out.println("Search method runtime (excluding user input): " + searchTimer.getElapsedTime() + " nanoseconds");
    System.out.println("Memory Used By Search: " + getMemoryUsedDuringMethod() + " bytes");
    System.out.println(" ");
    break;
```

Case 4 represents the option to search for a product in the database by its name. Before searching, the dataList is sorted using the mergeSort method. Then, the user is prompted to enter the product name to be searched. A Timer object searchTimer is created to measure the runtime of the search operation.

The program then performs a linear search on the sorted dataList to find the target product using the linearSearchByName method. If a product with the specified name is found (foundProductByName != null), the product details are printed. This includes the product name, price and quantity. If no product with the specified name is found, a message indicating the product is not found is displayed.

Then, the searchTimer is stopped, and the runtime and memory used during the searching operation are displayed to the user.

Case 5 (Exiting the program)

```
case 5:
    System.out.println("Exiting the program.");
    break;
```

Case 5 represents the option to exit the program. If the user chooses 5, the Scanner object will be closed and the program will be exited.

Default case

```
default:
    System.out.println("Invalid choice.");
    break;
```

If the user chooses anything other than 1-5, invalid choice will be shown in the console because there are no functions implemented for numbers other than 1-5.

METHODS

readDataFromFile

```
// Read data from the input file and store it in a LinkedList
private static LinkedList<Product> readDataFromFile(String filePath) throws IOException {
    // Create a new LinkedList to store Product objects read from the file
    LinkedList<Product> productList = new LinkedList<>();

    try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
        String line;

        // Read each line of the input file
        while ((line = reader.readLine()) != null) {
            // Split the line into separate parts using comma as the delimiter
            String[] parts = line.split(",");

            // Extract the product ID from the first part and parse it to an integer
            int id = Integer.parseInt(parts[0]);

            // Extract the product name from the second part
            String name = parts[1];

            // Remove the dollar sign and extract the product price from the third part and parse it to a double
            double price = Double.parseDouble(parts[2].replace("$", ""));

            // Extract the product quantity from the fourth part and parse it to an integer
            int quantity = Integer.parseInt(parts[3]);

            // Create a new Product object using the extracted data and add it to the LinkedList
            productList.add(new Product(id, name, price, quantity));
        }
    }

    // Return the LinkedList containing all the Product objects read from the file
    return productList;
}
```

This readDataFromFile method is responsible for reading data from the database text file and converting the data into a list of Product objects stored in a LinkedList.

First, the program creates an empty LinkedList<Product> called productList to store the Product objects. Then, a BufferedReader is used to read the data from the file. The program will read each line from the file, line by line, until there are no more lines left to be read. The readLine() method returns null when the end of the file is reached.

For every line read, it splits the line into individual parts using the comma (,) as the delimiter. This is done using the split(",") method, which returns an array of strings. The parts[] array now contains four elements, each representing the ID, name, price, and quantity of a product respectively. It gets the ID from the first element of the parts[] array, parts[0], using Integer.parseInt(parts[0]), the name from the second element, parts[1], and the price from the third element parts[2].

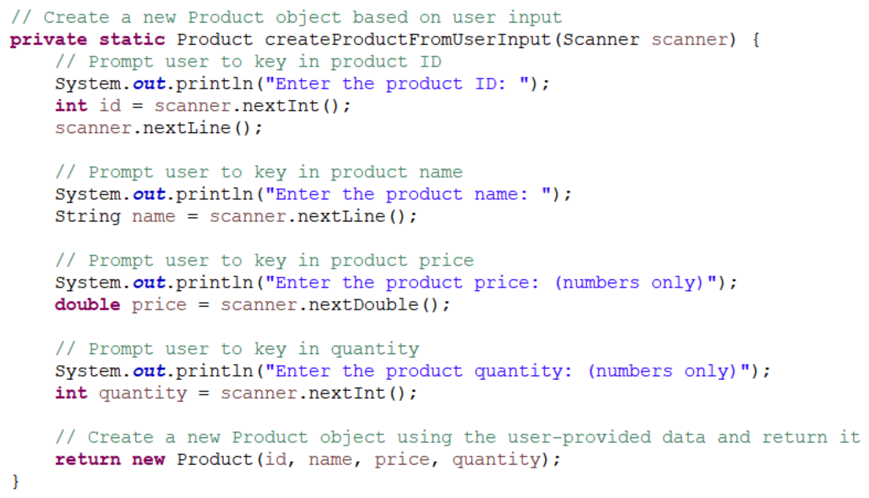
For the price, it is a bit different from the rest. The dollar sign ($) has to be removed first. The replace("$", "") method is used to remove the dollar sign ($) from the price string, and then Double.parseDouble() is used to convert the resulting string into a double value.

After that, it gets the quantity from the fourth element of the parts[] array, parts[3], using Integer.parseInt(parts[3]). With the extracted ID, name, price, and quantity, a new Product object is created and added to the productList using productList.add(new Product(id, name, price, quantity)).

The process will repeat for every line in the database text file. Once all lines are read, the method returns the list containing all the Product objects created from the data in the file.

To make it simple, the program reads a file, parses each line to get the necessary information, creates Product objects from these data, and stores them in a LinkedList.

createProductFromUserInput



```
// Create a new Product object based on user input
private static Product createProductFromUserInput(Scanner scanner) {
    // Prompt user to key in product ID
    System.out.println("Enter the product ID: ");
    int id = scanner.nextInt();
    scanner.nextLine();

    // Prompt user to key in product name
    System.out.println("Enter the product name: ");
    String name = scanner.nextLine();

    // Prompt user to key in product price
    System.out.println("Enter the product price: (numbers only)");
    double price = scanner.nextDouble();

    // Prompt user to key in quantity
    System.out.println("Enter the product quantity: (numbers only)");
    int quantity = scanner.nextInt();

    // Create a new Product object using the user-provided data and return it
    return new Product(id, name, price, quantity);
}
```

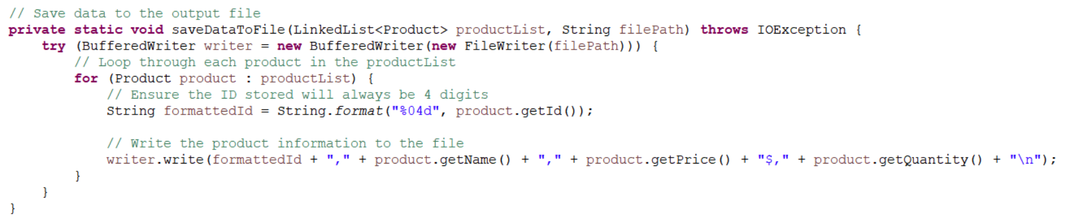
This createProductFromUserInput method is designed to interact with the user to gather information needed to create a new Product object. The Scanner class is used to read user input.

The method starts by prompting the user to enter the product ID using System.out.println("Enter the product ID: "). Then it reads an integer value using scanner.nextInt(). After reading the integer value, it consumes the newline character by using scanner.nextLine(). This is a necessary step because nextInt() does not consume the entire line.

The method then proceeds to prompt the user to enter the product name using System.out.println("Enter the product name: ") and reads the name as a string using scanner.nextLine(). Then, it prompts the user to enter the product price using System.out.println("Enter the product price: (numbers only)") and reads the price as a double using scanner.nextDouble(). Finally, it prompts the user to enter the product quantity using System.out.println("Enter the product quantity: (numbers only)") and reads the quantity as an integer using scanner.nextInt().

Once the method has collected all the product details, which includes ID, name, price, and quantity, it creates a new Product object using these values with the new Product(id, name, price, quantity) constructor and returns.

saveDataToFile



```java
// Save data to the output file
private static void saveDataToFile(LinkedList<Product> productList, String filePath) throws IOException {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(filePath))) {
        // Loop through each product in the productList
        for (Product product : productList) {
            // Ensure the ID stored will always be 4 digits
            String formattedId = String.format("%04d", product.getId());

            // Write the product information to the file
            writer.write(formattedId + "," + product.getName() + "," + product.getPrice() + "$," + product.getQuantity() + "\n");
        }
    }
}
```

This saveDataToFile method is created and used to take the productList, which is a LinkedList of Product objects, and the filePath, which is the filepath of the database text file as parameters. It uses a BufferedWriter to write data to the database text file.

The method starts by creating a BufferedWriter wrapped by FileWriter to write data to the database. It will then enter a loop to process every Product object in the productList.

For every Product object, it will ensure that the product ID will always be four digits by using String.format("%04d", product.getId()). It will then write the product ID, name, price together with the dollar ($) sign, and quantity to the database text file in a comma-separated format. The product information is written as a single line in the database text file. It will look like this: 0001, Toothbrush, 3.00$, 200.

After writing the product information for each product, the method uses a newline character \n to move to the new line in the database text file. The method continues this process until every product in the productList have been written to the database text file.

mergeSort

```java
// Merge Sort Algorithm based on Product IDs
private static void mergeSort(LinkedList<Product> dataList) {
    if (dataList.size() <= 1) {
        return; // The list is already sorted
    }

    // Split the list into two halves
    int mid = dataList.size() / 2;
    LinkedList<Product> left = new LinkedList<>(dataList.subList(0, mid));
    LinkedList<Product> right = new LinkedList<>(dataList.subList(mid, dataList.size()));

    // Recursively sort both halves
    mergeSort(left);
    mergeSort(right);

    // Merge the sorted halves
    merge(dataList, left, right);
}
```

This mergeSort method is created and implemented for sorting the dataList of Product objects based on their IDs using the Merge Sort algorithm.

Before starting the sorting process, the method will check if the dataList is already sorted or not. If the dataList has only one element or is empty, it means the list is already sorted, so the method will return without performing anything. If the list has more than one element, the method proceeds with the sorting process.

It calculates the middle index mid to split the dataList into left half and right half by using dataList.size()/2. Then, it creates two new LinkedList named left and right that represent the left and right halves of the original dataList. The method then recursively calls mergeSort on both left and right halves to sort them individually. Once the left and right halves are sorted, the method calls the merge method to merge the two sorted halves back into the original dataList.

merge

```
// Merge two sorted lists (left and right) into the main data list (dataList).
private static void merge(LinkedList<Product> dataList, LinkedList<Product> left, LinkedList<Product> right) {
    int leftIndex = 0; // Index for the left list
    int rightIndex = 0; // Index for the right list
    int dataIndex = 0; // Index for the main data list (dataList)

    // Compare elements from both left and right lists and merge them in sorted order
    while (leftIndex < left.size() && rightIndex < right.size()) {
        if (left.get(leftIndex).getId() < right.get(rightIndex).getId()) {
            // If the ID of the left element is smaller, add it to the main data list
            dataList.set(dataIndex++, left.get(leftIndex++));
        } else {
            // If the ID of the right element is smaller, add it to the main data list
            dataList.set(dataIndex++, right.get(rightIndex++));
        }
    }

    // Copy any remaining elements from the left and right lists to the main data list
    while (leftIndex < left.size()) {
        dataList.set(dataIndex++, left.get(leftIndex++));
    }
    while (rightIndex < right.size()) {
        dataList.set(dataIndex++, right.get(rightIndex++));
    }
}
```

This merge method is created and used for merging two sorted lists into the original dataList. This method uses three index variables: leftIndex, rightIndex, and dataIndex. These indexes are used to keep track of the current positions in the left list, right list, and the dataList.

The method uses a while loop to iterate through left and right lists, comparing the IDs of the elements at the corresponding indexes. During the merging process, the method ensures that the elements from both lists are merged into the dataList in ascending order based on their IDs.

For each iteration of the loop, the method compares the IDs of the elements at the current leftIndex and rightIndex. If the ID of the element in the left list is smaller, it means that that element should come first in the sorted order, so it is added to the dataList at the current dataIndex. And if the ID of the element in the right list is smaller, it is added to the dataList instead of the element in the left list. The index variables will be incremented after every loop. For example, if the element in the left list is added to the dataList, leftIndex and dataIndex will be incremented while rightIndex will be remained as the same. The process continues until all the elements from either the left or right list have been merged into the dataList.

Finally, if there are any remaining elements in either the left or right lists, they will be copied to the dataList using the remaining values of the index variables. After the merging process is complete, the dataList will contain the sorted elements from both the left and right lists in ascending order based on their IDs.

linearSearchById

```
// Linear search method to find a product by ID
private static int linearSearchById(LinkedList<Product> dataList, int target) {
    int index = 0;
    // Loop through each element in the dataList
    for (Product element : dataList) {
        // Compare the ID of the current element with the target
        if (element.getId() == target) {
            return index; // Target found, return the index where it was found
        }
        index++;
    }
    // Target not found, return -1
    return -1;
}
```

This linearSearchById method performs a linear search on the dataList to find a product that matches with the ID entered by the user. It uses a for-each loop to iterate through each product in the dataList.

During each iteration of the loop, the method compares the ID of the current element in the dataList with the ID entered by the user. If the ID of the element matches the ID entered, it means that the product that the user is looking for has been found in the list. The method then returns the index where the product was found. The index variable is used to keep track of the current position in the dataList.

If the ID entered by the user is not found in the dataList, the method returns -1 to indicate that the product is not present in the dataList.

linearSearchByName

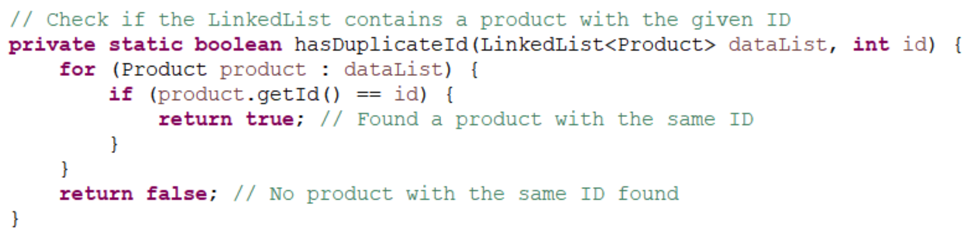
```
// Linear search method to find a product by name
private static Product linearSearchByName(LinkedList<Product> dataList, String targetName) {
    // Loop through each product in the dataList
    for (Product product : dataList) {
        // Search for a product with the name that matches the targetName, ignoring case
        if (product.getName().equalsIgnoreCase(targetName)) {
            return product; // Product found, return the product object
        }
    }
    // Product with the specified name not found, return null
    return null;
}
```

Similar to linearSearchById method, this linearSearchByName method performs a linear search on the dataList to find a product that matches with the ID entered by the user. It uses a for-each loop to iterate through each product in the dataList.

For each iteration of the loop, the method compares the name of the current product with the name entered by the user, ignoring the upper case and lower case by using equalsIgnoreCase. If the name of the product matches the named entered by the user, it means that the product that the user is looking for has been found in the list. The method then returns the product object representing the found product.

If no product with the specified name is found in the dataList, the method returns null to indicate that no product with the specified name exists in the list.

hasDuplicateId



```java
// Check if the LinkedList contains a product with the given ID
private static boolean hasDuplicateId(LinkedList<Product> dataList, int id) {
    for (Product product : dataList) {
        if (product.getId() == id) {
            return true; // Found a product with the same ID
        }
    }
    return false; // No product with the same ID found
}
```

If the user wants to add a product, this hasDuplicateId method checks whether there is already a product in the dataList with the same ID as the one entered by the user. It takes two parameters: dataList, which is the LinkedList containing the products, and id, the ID entered by the user to be checked. The method uses a for-each loop to iterate through each product in the dataList.

For each product, the method compares the ID of the current product with the ID entered by the user by calling the getId() method of the Product class. If it finds any product with the same ID, it means there are duplicates, and the method returns true. If the loop finishes without finding any product with the same ID, it means there are no duplicates, and the method returns false.

Product class

```
// Product class
private static class Product {
    private int id;
    private String name;
    private double price;
    private int quantity;

    // Constructor to create a new Product object with the given attributes
    public Product(int id, String name, double price, int quantity) {
        this.id = id;
        this.name = name;
        this.price = price;
        this.quantity = quantity;
    }

    // Getter method to retrieve the product's ID
    public int getId() {
        return id;
    }
    // Getter method to retrieve the product's name
    public String getName() {
        return name;
    }
    // Getter method to retrieve the product's price
    public double getPrice() {
        return price;
    }
    // Getter method to retrieve the product's quantity
    public int getQuantity() {
        return quantity;
    }
    @Override
    public String toString() {
        return "Product [ID=" + id + ", Name=" + name + ", Price=" + price + ", Quantity=" + quantity +"]";
    }
}
```

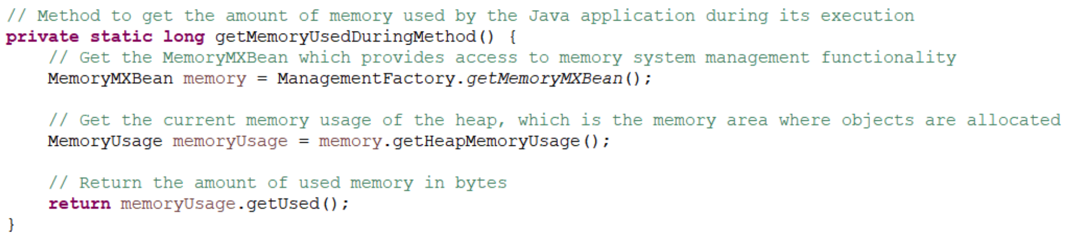
This Product class has four private variables, which are id, name, price, and quantity, representing the product ID, name, price, and quantity respectively.

The Product class contains a constructor, which is used to create a new Product object. The constructor takes four parameters, which are id, name, price, and quantity, and assigns them to the corresponding attributes.

The class provides getter methods, including getId(), getName(), getPrice(), and getQuantity() to retrieve the values of its attributes.

The toString() method is overridden to provide a customized string representation of the Product object. When the toString() method is called, it will return a string with the ID, name, price, and quantity of the product.

getMemoryUsedDuringMethod



```java
// Method to get the amount of memory used by the Java application during its execution
private static long getMemoryUsedDuringMethod() {
    // Get the MemoryMXBean which provides access to memory system management functionality
    MemoryMXBean memory = ManagementFactory.getMemoryMXBean();

    // Get the current memory usage of the heap, which is the memory area where objects are allocated
    MemoryUsage memoryUsage = memory.getHeapMemoryUsage();

    // Return the amount of used memory in bytes
    return memoryUsage.getUsed();
}
```

This getMemoryUsedDuringMethod() is used to measure the amount of memory used by the Java application during its execution. memory.getHeapMemoryUsage() returns the current memory usage of the heap, and memoryUsage.getUsed() returns the amount of used memory in bytes. This method returns the amount of memory used by the application during its execution.

4.1.3 DATABASE

Through to test the codes, we have been required to do 1000 of databases and store them into one text file. When the code runs, the database runs in the code and constructs an output that uses linked lists, linear search, and merge sort, or array lists, binary search, and bubble sort. Basically, we also face a lot of problems when construct a database, because the amount of information required is very huge. Then we have 4 sections in our database such as ID number, product name, price and product quantity. Therefore, for the ID number, we decided to use a 4-digit number from 1 to 1000. On the other hand, since we are doing supermarket management software, product names can also be found in supermarkets, such as Apple, bitter, soap, etc. We collect product names from several internets, since it is a huge database, so we search the internet and try to get the database. Fortunately, we did collect 1000 different product names and store them in the database After getting the product name, we managed to do the price for each product as well. We use the Java loop to insert a random price in the range of 5 to 105 for each product and add a $ after the price number, so this works and saves a lot of time by entering the prices one by one.

```
// Find the line to modify and add random prices
while ((line = reader.readLine()) != null) {
    String[] parts = line.split(", "); // Split into two parts: product code/name and price
    if (parts.length == 2) {
        double randomPrice = 5 + random.nextDouble() * 100; // Random price between 5 and 105
        String formattedPrice = String.format("$%.2f", randomPrice);
        content.append(parts[0]).append(" ").append(formattedPrice).append("\n");
        // Preserve the original product code/name and format the random price
    } else {
        content.append(line).append("\n"); // Append existing line with a new line character
    }
}
```

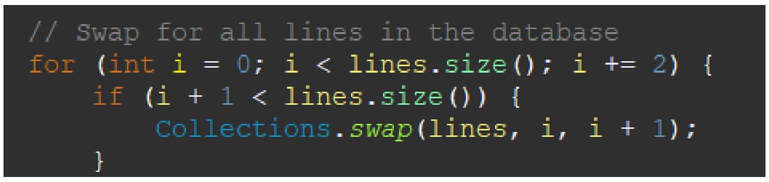
Also, we are including the quantity of the product in the database by using eclipse code, and the same concept as added the price number in database, also the quantity range we set it 30 to 498.

```
// Generate a random quantity between 30 and 498
int quantity = random.nextInt(469) + 30; // 469 is the range (498 - 30 + 1)

// Append the quantity to the line
String updatedLine = String.format("%s,%s,%s,%d", id, name, price, quantity);
updatedLines.add(updatedLine);
```

Moreover, we did run into some issues with spaces in the database before construct the final version of database. At first, we divided the information into 4 parts: ID, Name, Price, Quantity by adding spaces, but after reading our database in the code, the database cannot be run successfully due to the space problem in the text file. Therefore, we decided to separate the 4 parts with commas instead, but we couldn't add commas one by one because it would take longer. Therefore, we also use codes to separate the sections with commas.

```
// Reconstruct the line with comma and replace the space
String updatedLine = id + "," + productName + "," + price + "," + quantity;
contentBuilder.append(updatedLine).append("\n");
```

We made many changes to the database because of spacing issues as mentioned before and sorted issues. The sorting issue is due to the ID number of the product, because when construct the database, we set it as an ascending number which 0001 to 1000, but depending on our assignment, we need to sort the ID using bubble sort or merge sort. Therefore, we need test whether the code can successfully run to sort the database by ID number. By swapping the line, we use a for loop to swap all the rows in the database.



```
// Swap for all lines in the database
for (int i = 0; i < lines.size(); i += 2) {
    if (i + 1 < lines.size()) {
        Collections.swap(lines, i, i + 1);
    }
```

4.2 APPLICATION ANALYSIS AND COMPARISON

In the analysis and comparison, we will discuss the run time efficiency, memory usage of the code as well as the complexity of the code we used in Application 1 and Application 2. In the following discussion part, we will also be including the explanation of function that the algorithm and data structure that we used in our applications.

4.2.1 COMPARISON RUN TIME AND MEMORY USAGE

Data Structure + Algorithm	Operation	Run Time (ns)	Memory Usage (bytes)
ArrayList + Binary Search + Bubble Sort (Original Application 1)	Add ID	360,673,400	108,480,336
	Search ID	1,295,400	110,577,488
	Delete ID	8,837,100	110,577,488
	Add Name	248,873,200	87,177,632
	Search Name	288,000	103,388,048
	Delete Name	6,804,300	89,274,784
LinkedList + Linear Search + Merge Sort (Original Application 2)	Add ID	13,361,256,900	4,194,304
	Search ID	122,100	8,388,608
	Delete ID	13,415,200	8,388,608
	Add Name	36,321,088,800	4,194,304
	Search Name	107,400	8,388,608
	Delete Name	12,448,700	8,388,608

Table 1 comparison of both applications due to run time and memory usage

In this analysis, we will discuss about the run time and the memory usage of both applications. In application 1, it makes use of an arraylist data structure with a bubble sort for sorting. Also, it included binary search for ID operations. It reaches a run time of 360,673,400 nanoseconds and uses 108,480,336 bytes of memory while adding a new ID. On the other hand, it has a successful deletion which takes 978,500 nanoseconds while an unsuccessful deletion takes 8,837,100 nanoseconds to complete. In this case, the amount of memory usage for deletion stays the same at 110,577,488 bytes. Therefore, the application displays a binary search time of 1,295,400 nanoseconds and memory utilization of 110,577,488 bytes while looking for an existing ID. In addition, it also works well for adding product names which takes 248,873,200 nanoseconds to run and used of 87,177,632 bytes of memory. While deleting product names results in quick execution for deletions that are successful with a run time of 6,804,300 nanoseconds and delayed execution for deletions that are failed with a run time of 570,600 nanoseconds. In deletions for applications 1, the amount of memory used is still 89,274,784 bytes. When looking for already-used product names, it has a run time of 288,000 nanoseconds and utilizes103,388,048 bytes of memory when one is not discovered.

In application 2, it used a linkedlist data structure, merge sort for sorting and linear search for ID operations. In this application, adding a new ID result in slightly longer run times which takes a run time of 13,361,256,900 nanoseconds, but less memory use which takes 4,194,304 bytes. Moreover, a current ID can be deleted quickly with a run time of 13,415,200 nanoseconds, while only 8,388,608 bytes of memory are used. Yet, the run time for searching an existing ID, which uses 8,388,608 bytes of memory and is only 122,100 nanoseconds. In this application, adding product names has a longer run time of 36,321,088,800 nanoseconds and utilizes the same amount of memory which is 4,194,304 bytes as same as adding IDs. Nonetheless, when deleting product names, its successful remove takes only 12,448,700 nanoseconds to complete, whereas unsuccessful deletions require 8,388,608 bytes. Therefore, regardless of whether the linear search is successful, the product name search process runs faster than Application 1, requiring 8,388,608 bytes of memory in both cases and it takes 107,400 nanoseconds of run time.

Comparison between two applications due to the run time and memory usage

By comparing these two applications, when adding new IDs and product names, application 1 is runs faster than application 2. It is preferred in cases where a fast data insertion is required because of its efficient used of ArrayList, binary search, and bubble sort. Although Application 2 uses linked lists, linear search, and merge sort, it excels in terms of memory efficiency. Also, application 2 is better for those projects with a limited memory resources due to it uses less memory during the addition of IDs and product names.

Otherwise, application 1 shows its efficiency in speed while deleting the IDs and product names, it is providing a quicker operation compared to application 2. In contrast, application 2 performs well during unsuccessful delete operations due to highlight its value in certain circumference when repeated unsuccessful remove are anticipated. Also, both applications use almost same amount of memory usage during the deletion process, means there is not much different in their memory efficiency between both applications.

Next, by comparing the search time when IDs and product names are located, application 2 is better than application 1 in quickly searching ID and product name. However, Application 2 is still faster than Application 1 even though the search is unsuccessful. During search activity, again both applications use memory in a similar and it means that the memory efficiency is keeping a balance between both applications.

In a nutshell, the decision between both applications is dependent on the project's requirements. Therefore, application 1 implement with ArrayList, Binary Search, and Bubble Sort, it is a better choice if speed is crucial and memory use is not an issue. However, application 2 implement with LinkedList, Linear Search, and Merge Sort, so it would be more suitable for those projects prioritizing memory efficiency without compromising moderate speed.

4.2.2 COMPLEXITY OF EACH APPLICATION

Application 1: ArrayList with Binary Search and Bubble Sort

In this application, the text file database is managed and stored using an ArrayList. Furthermore, ArrayList supports constant time complexity (O (1)) for index-based element access. However, a linear search would require O(n) time complexity when it is looking into specific information in database. Also, we use Binary Search algorithm which runs in O (log n) time complexity, to speed up the search process efficiently.

On the other hand, to further sort the elements in the ArrayList according to their qualities, we decided to add Bubble Sort Algorithm in our code. For larger database, time complexity of bubble sort, O(n^2) makes it ineffective. Despite these flaws, bubble sort still a pretty simple algorithm to use.

Application 2: LinkedList with Linear Search and Merge Sort

In the second application, we manage the database read from the text file using a LinkedList. Because it necessitates traversing the list sequentially, the LinkedList requires linear time complexity (O(n)) for accessing elements as opposed to the ArrayList. Therefore, we use the Linear search method, which is less effective than Binary Search but adequate for unsorted data, to search the elements in the LinkedList. Yet, Linear Search operates in O(n) time complexity.

In addition, we use the Merge Sort technique to arrange the database objects in the LinkedList in descending order. In this application, Merge Sort is more effective than Bubble Sort for larger database because it has time complexity of O (n log n). Also, Merge Sort separates the list into smaller chunks and sorts each one separately, then merges the pieces back together in the original order of sorting.

Small Summarize

Therefore, when it comes to managing a database read from a text file, each application has their own set of advantages and challenges. In Application 1 uses an ArrayList with Bubble Sort and Binary Search to provide simple sorting and efficient searching. While, Application 2 uses a LinkedList with Linear Search and Merge Sort, which is a balance between search effectiveness and more complex sorting options.

Data Structure + Algorithm	Operation	Run Time (ns)	Memory Usage (bytes)
ArrayList + Binary Search + Bubble Sort (Original Application 1)	Add ID	360,673,400	108,480,336
	Search ID	1,295,400	110,577,488
	Delete ID	8,837,100	110,577,488
	Add Name	248,873,200	87,177,632
	Search Name	288,000	103,388,048
	Delete Name	6,804,300	89,274,784
ArrayList + Binary Search + Merge Sort	Add ID	13,673,000	8,388,608
	Search ID	1,056,800	8,388,608
	Delete ID	12,495,300	10,485,760
	Add Name	22,251,400	1,229,120
	Search Name	172,200	1,229,120
	Delete Name	4,175,500	1,229,120
ArrayList + Linear Search + Bubble Sort	Add ID	750,695,500	110,553,648
	Search ID	1,291,200	87,546,480
	Delete ID	10,021,400	112,650,000
	Add Name	252,492,400	87,230,256
	Search Name	579,200	87,230,256
	Delete Name	6,376,200	87,230,256
LinkedList + Linear Search + Merge Sort (Original Application 2)	Add ID	13,361,256,900	4,194,304
	Search ID	122,100	8,388,608
	Delete ID	13,415,200	8,388,608
	Add Name	36,321,088,800	4,194,304
	Search Name	107,400	8,388,608
	Delete Name	12,448,700	8,388,608
Linked list + Binary Search + Bubble sort	Add ID	3,266,692,900	4,194,304
	Search ID	149,700	4,194,304
	Delete ID	1,136,400	4,194,304
	Add Name	3,893,900	0
	Search Name	180,800	64,229,520
	Delete Name	1,844,700	64,229,520
Linked list + Binary Search + Merge sort	Add ID	5,038,610,600	4,194,304
	Search ID	135,800	8,388,608
	Delete ID	1,060,500	8,388,608
	Add Name	7,224,698,300	4,194,304
	Search Name	182,300	8,388,608
	Delete Name	1,006,800	8,388,608

Table 2 Comparison the run time and memory usage of data structures with different Algorithms

1. ***Comparison efficiency of Sorting between ArrayList +binary search +bubble sort and ArrayList + binary search + merge sort***

In this session, we may compare the effectiveness of sorted between ArrayList + Binary Search + Bubble Sort and ArrayList + Binary Search + Merge Sort as shown in Table 2.

First, it is clear from looking at the run time that ArrayList + Binary Search + Merge Sort performs better than ArrayList + Binary Search + Bubble Sort for the majority of operations. For instance, Merge Sort significantly outperforms Bubble Sort in terms of Add ID, Search ID, and Delete ID execution times. This shows that the Merge Sort implementation is quicker and more effective in finding, adding, and removing members from the ArrayList. On the other hand, both algorithms perform similarly in terms of memory utilization, with the majority of operations showing just modest variations in memory consumption. Therefore, neither approach has a considerable memory use advantage.

According to these findings that we found, ArrayList + Binary Search + Merge Sort proves to be a faster solution for the majority of operations. The reliability of the sorting process, the quantity of the dataset, and the ease of implementation should all be taken into account when choosing the right algorithm.

2. ***Comparison efficiency of searching between linked list + linear search + merge sort and Linkedlist + binary search + merge sort***

According to the Table 2, in this comparison we are aimed to compare the search efficiency and the memory usage of both of LinkedList + Linear Search + Merge Sort and LinkedList + Binary Search + Merge Sort.

The run time results show that for most operation which are LinkedList + Linear Search + Merge Sort performs better than LinkedList + Binary Search + Merge Sort. When compared to Binary Search + Merge Sort, the Add ID, Add Name, Search Name, and Delete Name actions with Linear Search + Merge Sort run much faster. However, for Search ID and Delete ID actions, Binary Search + Merge Sort performs faster. Moreover, both algorithms perform similarly in terms of memory usage, with the majority of operations showing just slight variations in memory consumption.

The results that we can see that LinkedList + Linear Search + Merge Sort looks to be more time efficient than LinkedList + Binary Search + Merge Sort for the majority of operations. Also, the size of the database, the frequency of operations, and the difficulty of the search procedure should all be considered while selecting the best method.

3. ***Comparison of data structure between ArrayList + Binary Search + Bubble Sort and Linked list + Binary Search + Bubble Sort***

Lastly, we compare the different data structure with same algorithm, as we can see that ArrayList + Binary Search + Bubble Sort and Linked list + Binary Search + Bubble Sort according to Table 2. The two data structures and algorithms may be easily distinguished after examining the run time and memory usage for each operation.

In addition, Linked list + Binary Search + Bubble Sort is more effective than ArrayList + Binary Search + Bubble Sort in terms of time performance which run the code by eclipse. Next, with the installation of linked lists, the time required for operations like Add ID, Search ID, and Delete ID is significantly reduced. This shows that for these particular workloads, the Linked list data structure paired with Binary Search and Bubble Sort has given a quicker execution speeds. Furthermore, both systems perform similarly in terms of memory usage, with the majority of operations showing just slight variations. The Add Name and Search Name actions particularly highlight the modest memory consumption advantage of Linked list + Binary Search + Bubble Sort.

Last but not least, these comparisons show that Linked list + Binary Search + Bubble Sort is a good option for the majority of operations due to its quick execution times and reasonable memory use. When deciding on the best data structure and method for a particular use case, it is crucial to take other aspects into account, such as the stability of the sorting process, the size of the dataset, and the ease of implementation.

4.2.3 COMPARISON COMPLEXITY BETWEEN TWO APPLICATIONS

In this comparison analysis, we will assess into the strengths and weaknesses of two applications that used two data structures and sorting algorithm. Thus, Application 1 uses an ArrayList with Binary search and Bubble sort while Application 2 used a LinkedList with Linear Search and Merge Sort. The objective of this analysis is to determine which application is best suitable choices as shown in Table 3 below for particular use cases, taking into considering each application's efficiency and their performance.

Data Structure + Algorithm	Search Time Complexity	Sort Time Complexity	Memory Complexity
ArrayList + Binary Search + Bubble Sort (Application 1)	O(log n)	O(n^2)	O(n)
LinkedList + Linear Search + Merge Sort (Application 2)	O(n)	O(n log n)	O(n)

Table 3 Comparison complexity between both applications

Application 1: ArrayList with Binary Search and Bubble Sort

In Application 1, it takes use of the effectiveness of binary search by consider quick searches on a sorted ArrayList with a time complexity of O (log n). This makes it perfect for searching the information, especially when the database is limited. Also, adding to its appeal is the O (1) which is constant time access complexity of an ArrayList elements retrieved using an index.

However, the use of bubble sort cause Application 1 to perform slightly low performance in sorting. As we know when it comes to large database, bubble sort will become extremely inefficient with a complexity of O(n^2) during sorting operations. Therefore, this scenario that strongly rely on sifting through large amounts of data are not suitable for this application.

Application 2: LinkedList with Linear Search and Merge Sort

In Application 2, it makes use of Merge Sort's effectiveness, which operates effectively with a sorting time complexity of O (n log n). Therefore, it performs better than Bubble Sort in applications that require sorting larger database, giving it a significant performance advantage.

On the other hand, in terms of memory performance, using LinkedLists in Application 2 has the advantages, especially when handling frequent insertions and deletions. However, this advantage does come at the cost of less efficient search results. In addition, Linear Search algorithm has an O(n) time complexity, which can make it slower than Binary Search, especially for longer lists. Furthermore, the O(n) access time complexity of LinkedLists for element retrieval by index may restrict some operations.

Small Summarize

To summarize this, the comparison between Application 1 and Application 2 also has their advantages and disadvantages. In Application 1 which are ArrayList + Binary Search + Bubble Sort, due to its quicker search times and constant-time access, it can be a good option if the application mainly emphasizes searching operations on a limited database. However, in Application 2 which are LinkedList + Linear Search + Merge Sort is more effective when sorting larger database since it offers faster performance and better memory management through LinkedLists.

Other than that, in the previous session, we compare one by one such as ArrayList and LinkedList, Binary search and Linear search and so on. Instead of comparing one by one between application 1 and application 2, here also will include the comparison of complexity as shown in Table 4 and compare between each algorithm with other data structures.

COMPARISON WITH ALGORITHM AND DIFFERENT DATA STRUCTURE

Data Structure + Algorithm	Time Complexity	Space Complexity
ArrayList + Linear Search	O(n)	O(1)
LinkedList + Linear Search	O(n)	O(1)
Binary Search + LinkedList	O(log n)	O(1)
Binary Search + ArrayList	O(log n)	O(1)
Bubble Sort + LinkedList	O(n^2)	O(1)
Bubble Sort + ArrayList	O(n^2)	O(1)
LinkedList + Merge Sort	O(n log n)	O(n)
ArrayList + Merge Sort	O(n log n)	O(n)
Binary Search + Bubble Sort	O(n^2)	O(1)
Binary Search + Merge Sort	O(n log n)	O(1)
Linear Search + Bubble Sort	O(n^2)	O(1)
Linear Search + Merge Sort	O(n log n)	O(n)

Table 3 Comparison of complexity

1. ***Array list + Linear search and Linked list + Linear search***

When conducting a linear search, the widely used data structures ArrayList and LinkedList both exhibit an O(n) time complexity. Both data structures for linear search require traversing the full list to find the requested member, which adding time complexity proportionate to the list size. Thus, in this comparison both ArrayList and LinkedList exhibit comparable linear search performance.

However, when it considering additional procedures, their difference between them become apparent. ArrayList is a useful option for retrieving random elements because it supports constant-time access by index. In contrast, when given a reference to the node, LinkedList excels at handling insertions and removals at random points with an O(1) time complexity. Due to the additional reference needed in each node, LinkedList uses somewhat more memory per element than ArrayList, whose memory organization is simpler. It is crucial to assess the application-specific needs while deciding between the two data structures, giving priority to elements like access pattern, insertions, deletions, and memory limitations.

2. ***Binary search + Linked list and Binary search + Array list***

In this case, for searching in a sorted list, binary search is effective with a time complexity of O(log n). The overall performance is subpar when paired with LinkedList, which does not support constant-time access by index. Yet, the linear time complexity introduced by traversing the LinkedList results in a total time complexity of O(n log n). Moreover, binary search using LinkedList is less appropriate for handling huge datasets than binary search using ArrayList, which provides constant-time access by index.

On the other hand, eith sorted arrays like ArrayList, binary search performs well and has an O(log n) time complexity. The efficient random access requirement of Binary search is complemented by ArrayList's constant-time access by index, resulting in a speedier total search operation. Binary search using ArrayList is the method of choice for effective searching in large datasets, with a logarithmic time complexity.

In comparing both, binary search with LinkedList may have a combined time complexity of O(n log n) for large database, which makes it less effective. The best option for effective searching is binary search with ArrayList since it maintains O(log n) time complexity and is better suited for quick retrieval from sorted lists.

3. Bubble sort + Linked list and Bubble sort + Array List

In this comparison, Bubble sort is not effective for sorting huge datasets since it has an O(n2) time complexity when used with LinkedList. Also, there are multiple traversals of the list are necessary to swap elements by switching node links, which results in subpar performance. Although LinkedList provides quick insertion and deletion, Bubble sort's quadratic time complexity outweighs these benefits. Therefore, there is some small advised to utilize Merge sort or Quick sort, which are more effective sorting algorithms, for huge datasets.

Additionally, bubble sort with ArrayList has an O(n2) time complexity, making it unsuitable for sorting lengthy lists. Although ArrayList offers constant-time access, the sorting procedure requires numerous comparisons and swaps, which leads to poor performance as the collection expands. Also, it is preferable to use algorithms like Merge sort or Quick sort for sorting operations on ArrayLists since they have better time complexity (O(n log n)) and are more effective for processing huge datasets.

In comparing both, bubble sorting large datasets is not recommended because bubble sorting with LinkedList and bubble sorting with ArrayList both have quadratic temporal complexity.

4. Linked list + Merge sort and Array list + Merge sort

In this session, Merge Sort is a divide and rule algorithm with an O(n log n) time complexity for sorting, and LinkedList is a node-based data structure. By reordering node references during the merging stage, Merge Sort are more quickly handles sorting linked lists. Thus, due to its optimal time complexity, LinkedList with Merge Sort is a fantastic option for quickly sorting huge datasets.

Next, a dynamic array that offers constant time access to elements by index is called an Arraylist. The combination results in an effective sorting procedure when used with Merge Sort, which has a time complexity of O(n log n) for sorting arrays. Furthermore, merge sort merging step is enhanced by the constant-time access in ArrayList, which does not require expensive element shifting. Since random access is required for efficient sorting of arrays or lists, ArrayList with Merge Sort is a great choice, especially for huge datasets.

In comparison of both, LinkedList with Merge Sort and ArrayList with Merge Sort both provide effective sorting with an O(n log n) time complexity. Due to its effective merging through node references, LinkedList with Merge Sort is preferred for sorting linked lists, whereas ArrayList with Merge Sort is the best option for arrays or lists with constant-time access requirements.

5. Binary search + Bubble sort and Binary search + Merge sort

When used on a sorted list, binary search is a very effective searching method with a time complexity of O(log n). However, how well it performs is largely dependent on the sorting technique that was used to organize the data before to the search. Therefore, this meaning that the overall time complexity for the combined operations becomes O(n2) when combined with Bubble sort, a straightforward sorting algorithm with an O(n2) time complexity in both the worst and average scenarios. This is due to the fact that Bubble sort must be completed before Binary search can successfully locate the target element.Thus, Binary search with Bubble Sort is not advised for searching in large datasets since it performs sub-optimally due to Bubble Sort's quadratic time complexity, which exceeds the advantages of Binary search's logarithmic time complexity.

However, when Binary search is combined with Merge sort, a divide and rule algorithm with an overall time complexity of O(n log n), the combined operations have a time complexity of O(n log n). Despite the initial overhead of completing the Merge sort, Binary search prevails due to its logarithmic time complexity, making it much faster and more effective than Bubble sort. On the other hand ,for searching in huge datasets, binary search with merge sort is strongly advised because it offers a noticeable performance boost and enables effective retrieval of elements from sorted lists.

The comparison between both are while binary search is an effective searching method, the choice of the underlying sorting algorithm can have a big impact on how well it performs. In comparison to binary search with merge sort, which has a combined time complexity of O(n log n) and is the ideal method for effective searching in these situations, binary search with bubble sort has a combined time complexity of O(n2) and is not appropriate for huge datasets.

6. Linear search + Bubble sort and Linear search + Merge Sort

In this both comparisons, when searching a list sequentially, linear search has an O(n) time complexity. However, the total performance degrades when paired with Bubble Sort, which has an O(n2) time complexity. The combined operation of Bubble Sort has a quadratic time complexity of O(n2) since the sorting process is independent of the search operation. Therefore, Linear search using Bubble Sort is therefore not practicable for large database. For greater performance, other search algorithms like binary search or linear search with a more effective sorting method should be chosen.

In order to discover the target element, linear search systematically looks through the list with an O(n) time complexity. The overall performance is improved when paired with Merge Sort, which has a time complexity of O(n log n). Prior to the linear search, Merge Sort effectively sorts the list, with no impact on the linear search's time complexity. With an overall time complexity of O(n log n), linear search with merge sort is a good option for searching through huge datasets and performs better than linear search with bubble sort.

In this comparison we can see that, Linear search using Bubble Sort is inefficient for large datasets and has a quadratic time complexity (O(n2)). For effective searching in longer lists, however, Linear search with Merge Sort is a superior solution due to its more favourable time complexity (O(n log n)). It is critical to take the sorting method into account while conducting linear searches because it has a substantial impact on overall performance, particularly for bigger datasets.

CHAPTER V: CONCLUSION

5.1 CONCLUSION

In conclusion, according to the comparison above, we suggest using applications 2 which is LinkedList + Linear Search + Merge Sort **(highlighted in Table 2)** due to large database with 1000 data consists of IDs, product names, prices, and quantities. By selecting application 2, it can be managing a large number of databases. Since, it has better memory management use with the LinkedList data structure which is useful when working with such a large database. Additionally, it also enhancing the application's efficiency with the linear search and merge sort algorithm. Yet, it allows efficiency search and insertion times while conserving memory. In general, Application 2 can manage to handle large database such as IDs, product names, prices, and quantities because it is good at managing memory efficiency without effect other features such as search and sorting. The reason we choose application is also suitable to the goals of this project because it is testing the code efficiency by sorting, searching the information in the database, and it guarantees that database operations can be performed well without increasing the memory capacity of the system.

5.2 REAL LIFE APPLICATION

A software program called a supermarket management system automates several operations in a business. This method makes it possible to manage retail information effectively. Its applications could be numerous and important. Here are some examples of actual applications for supermarket management systems:

1. **POS Integration:**
   The system frequently integrates with the POS system of the supermarket, allowing for quick and accurate checkout procedures, real-time inventory adjustments, and simple payment processing.
2. **Inventory management:**
   The system keeps track of product stock levels, controls expiration dates, and automatically places orders when levels drop below a predetermined level. This keeps in-demand items from running out or being overstocked.
3. **Sales and Promotions:**
   For sales and promotions, the manager can plan and carry out sales events and promotions with the system's assistance. It makes it possible to monitor and evaluate the effectiveness of marketing.
4. **Supplier and Vendor Management:**
   In addition to managing a database of suppliers and vendors, the system can be used to manage deliveries, payments, and other transactions.
5. **Reporting and Analytics:**
   Supermarket management systems generate in-depth analyses and reports on a range of shop performance measures, including sales trends, top products, and profit margins. Making informed business decisions is aided by this information.

6. **Price Management:**

Depending on demand, seasonality, or competitor prices, the system can automate pricing strategies, update prices, and handle price revisions for various items.

7. **Shelf and Space Management:**
   It helps optimize product placement on shelves and controls the supermarket's overall layout to enhance customers' shopping experiences.

In a nutshell, a successful supermarket management system could increase operational effectiveness, client satisfaction, and financial success for supermarket owners and operators.

References:

1. Arraylist in Java (2023) GeeksforGeeks. https://www.geeksforgeeks.org/arraylist-in-java/

2. Arraylist vs LinkedList in Java (2023) GeeksforGeeks. https://www.geeksforgeeks.org/arraylist-vs-linkedlist-java/

3. Binary search - data structure and algorithm tutorials (2023) GeeksforGeeks. https://www.geeksforgeeks.org/binary-search/

4. baeldung, W. by: (2022) Time complexity of java collections, Baeldung. https://www.baeldung.com/java-collections-complexity

5. Can binary search be applied in an unsorted array? (2023) GeeksforGeeks. https://www.geeksforgeeks.org/can-binary-search-be-applied-in-an-unsorted-array/

6. Difference between arraylist and LinkedList - javatpoint (no date) www.javatpoint.com. https://www.javatpoint.com/difference-between-arraylist-and-linkedlist

7. Khandelwal, V. (2023) *What is merge sort algorithm: How does it work and its implementation: Simplilearn, Simplilearn.com*. https://www.simplilearn.com/tutorials/data-structure-tutorial/merge-sort-algorithm.

8. LinkedList in Java (2023) GeeksforGeeks. https://www.geeksforgeeks.org/linked-list-in-java/

9. Merge sort - data structure and algorithms tutorials (2023) GeeksforGeeks. https://www.geeksforgeeks.org/merge-sort/

10. Mishra, A. (2022) Performance analysis of ArrayList and LinkedList in Java - DZone, dzone.com.https://dzone.com/articles/performance-analysis-of-arraylist-and-linkedlist-i

11. Reeves, O. (2008) Sorting algorithms: The bubble sort, OJ's Perspective. https://buffered.io/posts/sorting-algorithms-the-bubble-sort/

12. Sharma, V. (2022) *Difference between linear search and binary search*, *Scaler Topics*. https://www.scaler.com/topics/linear-search-and-binary-search/

13. S, R.A. (2023) *What is Linear Search Algorithm: Time complexity*, *Simplilearn.com*. https://www.simplilearn.com/tutorials/data-structure-tutorial/linear-search-algorithm

14. Terra, J. (2023) Binary search algorithms: Overview, when to use, and examples: Simplilearn, Simplilearn.com. https://www.simplilearn.com/binary-search-algorithm-article

15. Tutu, A. (2016) Sorting algorithms: The difference between bubble sort and merge sort, Medium.https://codeamt.medium.com/sorting-algorithms-the-difference-between-bubble-sort-and-merge-sort-complexity.

16. Thyagharajon, A. (2014) Differences between merge sort and bubble sort - durofy - business, technology, entertainment and Lifestyle Magazine, Durofy. https://durofy.com/differences-between-merge-sort-bubble-sort

17. Time and space complexity analysis of Bubble Sort (2023) GeeksforGeeks. https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-bubble-sort/

18. Woltmann, S. (2022) Bubble sort – algorithm, source code, Time Complexity, HappyCoders.eu. https://www.happycoders.eu/algorithms/bubble-sort