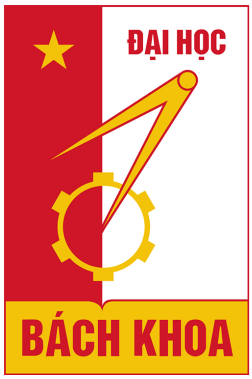
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY



OBJECT-ORIENTED PROGRAMMING - IT3100E

DATA STRUCTURE VISUALIZATION APP

Instructor: PhD. Tran The Hung
Students: Tran Cao Phong - 20226061
Dang Van Nhan - 20225990
Bui Ha My - 20225987
Tran Kim Cuong - 20226017

Ha Noi, June 2024

Contents

Abstract

Data structures are fundamental for efficient data management and algorithm performance. This report details the design and development of a Java application that simulates operations on essential data structures: stack, queue, and list. The application uses object-oriented programming to model these structures and features a graphical user interface (GUI) for intuitive interaction. Users can perform operations such as push, pop, enqueue, dequeue, add, and remove to visualize the results in real-time. Additionally, the app integrates artificial intelligence, allowing users to interact with a chatbot to gain a deeper understanding of these data structures within the scope of this project. This tool aims to enhance understanding and accessibility of data structure concepts through interactive learning.

1 Introduction

Data structures are an integral part of computer science and programming, providing the means to store, manage, and organize data efficiently. They are essential for the development of algorithms that need to perform tasks like searching, sorting, and data manipulation in a fast and efficient manner. The proper use of data structures can significantly enhance the performance of software applications, making them more responsive and capable of handling large volumes of data. This report delves into the importance of data structures and provides an in-depth analysis of three fundamental types: queues, stacks, and lists in Java. We will also use object-oriented programming (OOP) techniques to simulate operations on stacks, queues, and lists, and build a graphical user interface (GUI) to create an intuitive and user-friendly interface. Moreover, we will integrate to the app a chatbot powered by AI to help users gain deeper insights into these data structures.

In this section, we will introduce the theoretical basis of three data structures: stack, queue, and list, as well as their implementation in Java Collections Framework.

1.1 Queue

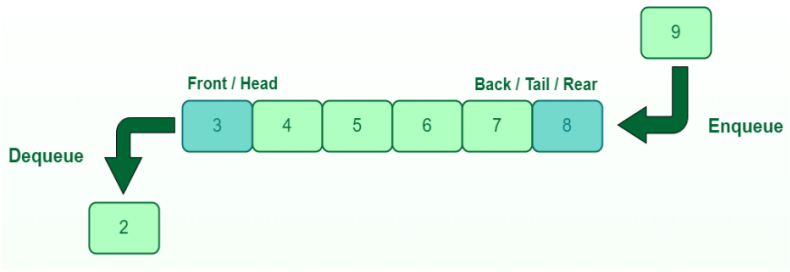


Figure 1: Queue (Source: GeeksforGeeks)

A queue is a fundamental data structure in computer science that operates on the First In, First Out (FIFO) principle. This means that the first element added to the queue is the first one to be removed. In Java, queues are part of the Java Collections Framework and are defined

by the `Queue` interface, which extends the `Collection` interface. Common implementations of the `Queue` interface include `LinkedList`, `PriorityQueue`, and `ArrayDeque`. Queues are widely used in various applications, such as task scheduling, managing requests in web servers, and breadth-first search algorithms. They support essential operations like `enqueue` (adding elements), `dequeue` (removing elements), `peek` (viewing the first element without removing it), and checking if the queue is empty. The efficient handling of these operations makes queues an indispensable data structure for ensuring orderly processing of elements in many software applications.

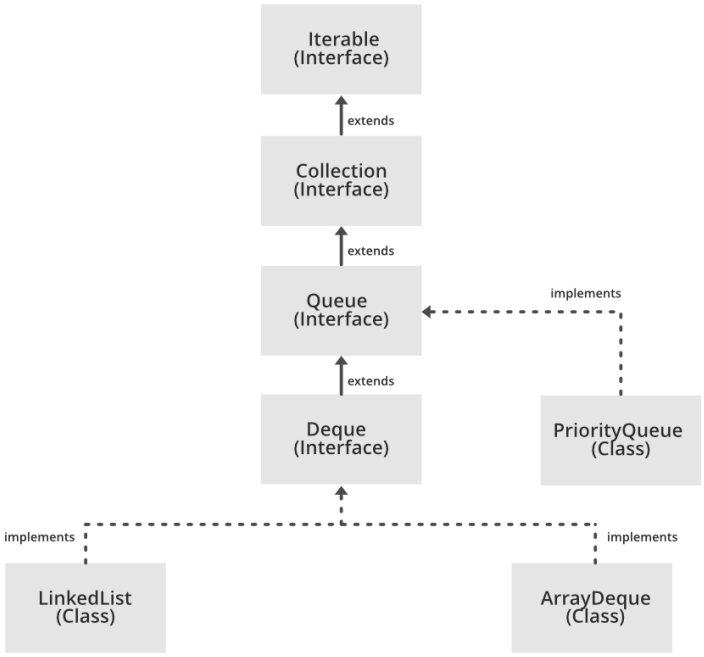


Figure 2: Queue Interface in Java (Source: GeeksforGeeks)

1.2 Stack

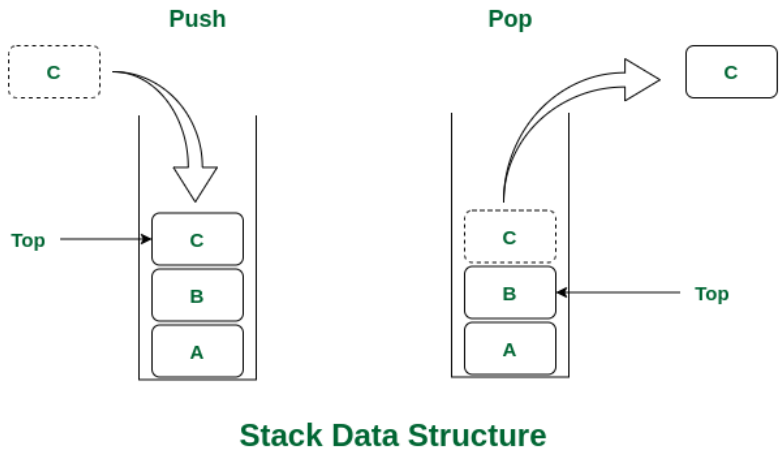


Figure 3: Stack (Source: GeeksforGeeks)

Stack is a data structure that operates on the Last In, First Out (LIFO) principle. This means that the last element added to the stack is the first one to be removed. Stacks are used in various applications, such as managing function calls, parsing expressions, and backtracking algorithms. In Java, stacks are represented by the `Stack` class, which is part of the Java Collections Framework. The `Stack` class extends `Vector` and provides standard stack operations such as `push` (adding an element to the top of the stack), `pop` (removing the top element), `peek` (viewing the top element without removing it), and `isEmpty` (checking if the stack is empty). The ease of use and the efficient handling of these operations make stacks a versatile and essential data structure in software development.

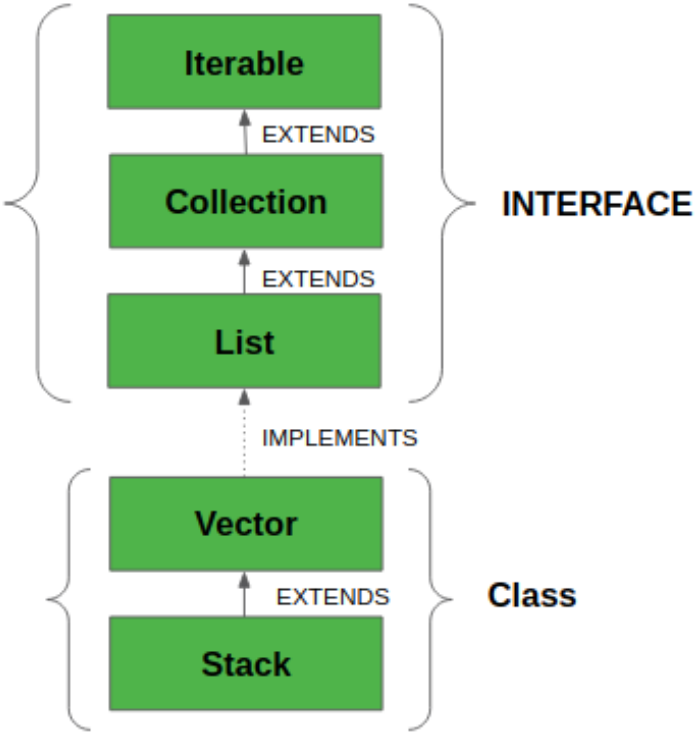


Figure 4: Stack In Java (Source: GeeksforGeeks)

1.3 List

List is a versatile data structure in computer science that represents an ordered collection of elements, allowing for dynamic resizing and providing methods to access, add, remove, and modify elements. In Java, the `List` interface, part of the Java Collections Framework, defines the operations of a list. This interface is implemented by several classes, including `ArrayList`, `LinkedList`, and `Vector`, each offering different performance characteristics. The `ArrayList` provides fast random access to elements but slow insertion and deletion in the middle of the list, while the `LinkedList` allows for efficient insertion and deletion at any position but slower access time. Lists are widely used in software development for tasks such as managing collections of objects, implementing other data structures, and facilitating algorithms that require sequential access to elements. The ability to handle a dynamic set of elements efficiently makes lists an indispensable tool in programming.

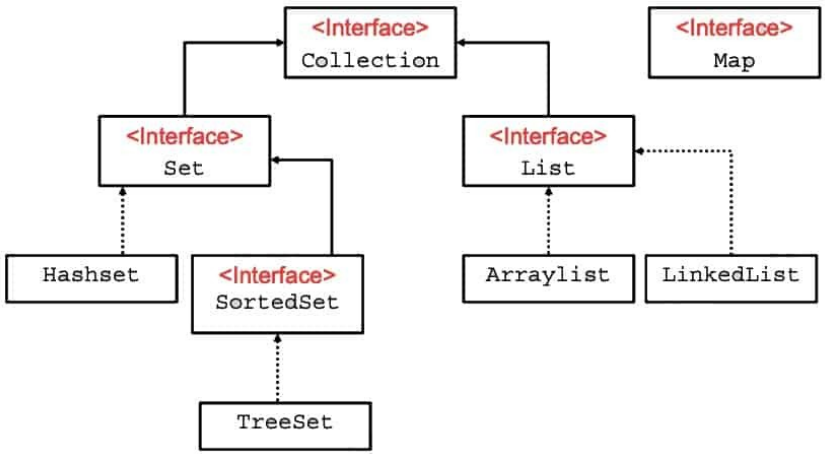


Figure 5: List In Java (Source: LinkedIn)

2 Use case diagram

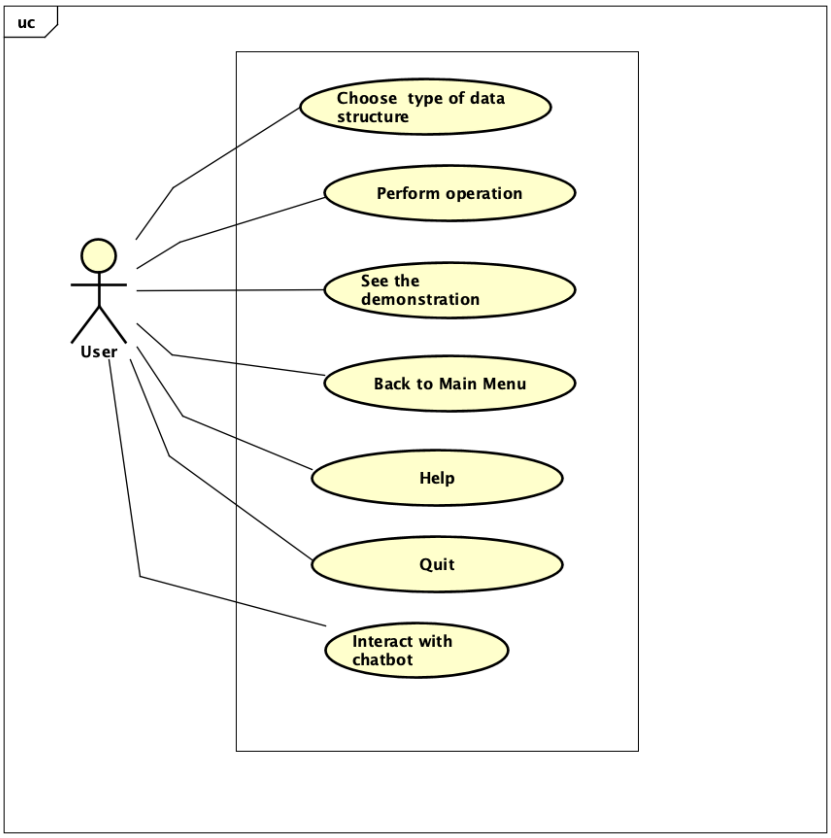


Figure 6: Use case diagram

With the requirement to create an application that simulates operations on data structures, our group has created a use case diagram with seven main functions:

- **Choose type of data structure:** This function allows the user to choose between queue, stack, or list to perform operations.

- **Perform operations:** This function allows the user to choose operations in queue/stack-/list, such as insert, delete,...
- **See demonstration:** This function is used to simulate the operations selected by the user and display them on the screen.
- **Back to Main Menu:** The user will use this function to return to the main menu.
- **Help:** The user will use this function to gain more knowledge about how to use this application.
- **Quit:** This function is used to quit the application when the user no longer wants to use it.
- **Interact with chatbot:** The user will choose this function to interact with chatbot powered by AI to gain more insights about the data structures.

3 General class diagram

In this section, our group develops a general class diagram to implement the project, with four packages, include three main packages, based on the **Model-View-Controller (MVC)** architecture. This design pattern was chosen for its ability to separate concerns, enhance code maintainability, and improve the scalability of the application.

Model-View-Controller (MVC) framework is a design pattern that divides an application into three core components: **Model**, **View**, and **Controller**. Each component addresses distinct development aspects, separating business logic from the presentation layer. Originally used for desktop GUIs, **MVC** is now a widely adopted framework for web and mobile app development, known for its scalability and extensibility. Created by Trygve Reenskaug, **MVC** aims to manage complex data by breaking down large applications into specific, purpose-driven sections.

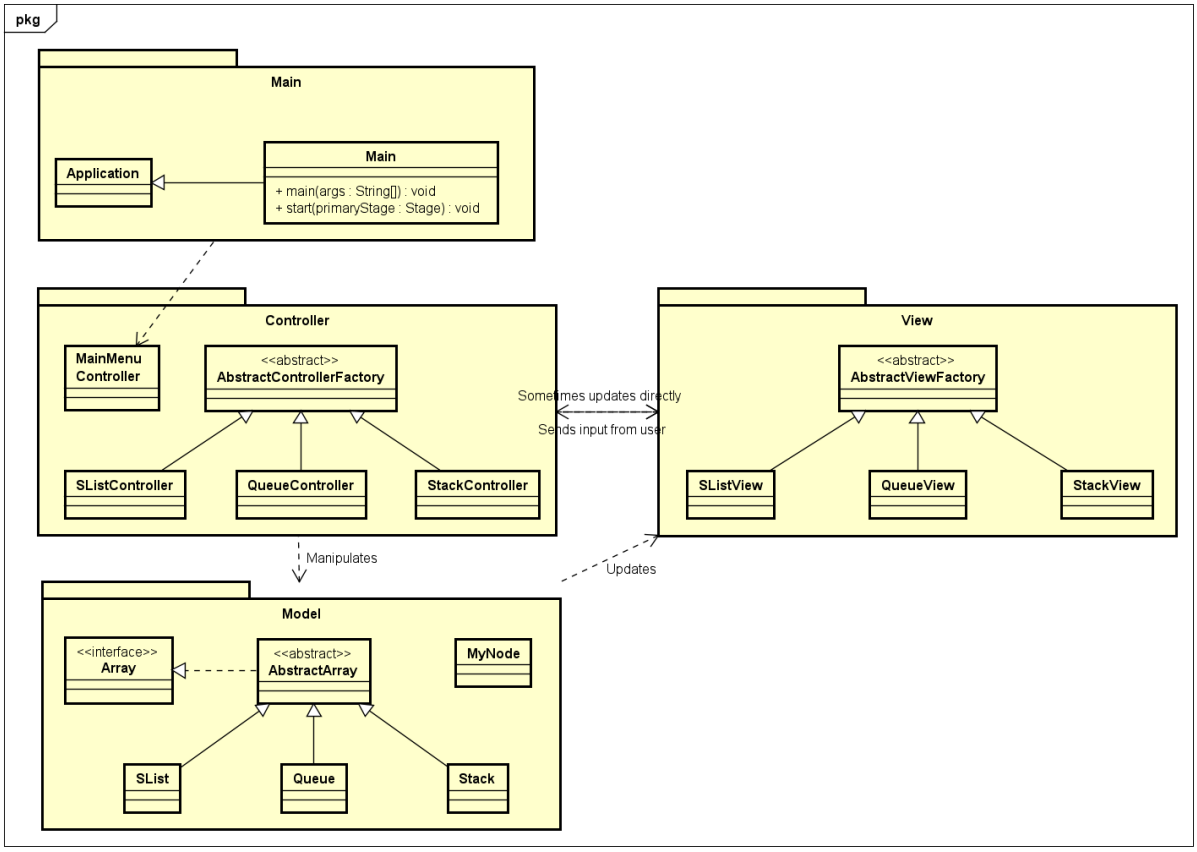


Figure 7: General Class Diagram

From the general class diagram, we can see that the **Model**, **View** and **Controller** packages work together to create a well-structured and maintainable application. The **Model** package is responsible for the core business logic and data management, encapsulating the data structures and operations on them, such as stack, queue, and list implementations. The **View** package handles the graphical user interface (GUI), presenting data to the user and capturing user interactions. It ensures that users have an intuitive and interactive experience while using the application. The **Controller** package acts as an intermediary between the **Model** and **View** packages. It processes user inputs received from the **View**, updates the **Model** accordingly, and then refreshes the **View** to reflect any changes in the data.

3.1 Main package

This package stores `main` class, cooperates with `MainMenuController` in **Controller** package to start our application.

3.2 Model package

The **Model** package stores the essential business logic and data handling capabilities of the application: `Array`, `AbstractArray`, `SList`, `Queue`, `Stack`, `MyNode`.

- `Array` interface defines a set of operations for array-like data structures. This interface ensures that any class implementing it will provide specific functionalities related to arrays. This interface is implemented by the `AbstractArray` class.

- `AbstractArray` is the abstract class that provides a partial implementation of the `Array` interface. It likely contains shared methods and properties that are common across various array-based data structures. This is the superclass of `SList`, `Queue`, `Stack` class and implements the `Array` interface.
- `Queue` is the class represents the queue data structure. This class extends `AbstractArray`, inheriting its functionalities and adding queue-specific operations such as enqueue and dequeue.
- `Stack` represents the stack data structure. This class extends `AbstractArray` and includes methods specific to stack operations like push and pop.
- `SList` represents a singly linked list data structure. This class extends `AbstractArray`, leveraging shared functionalities while implementing list-specific operations.
- `MyNode` class represents a node in a linked data structure. It typically contains data and a reference to the next node. This class is essential for implementing linked lists and other node-based structures. It is used by `SList`, `Queue`, and `Stack` to manage elements within these data structures.

3.3 View package

The **View** package is responsible for handling the graphical user interface (GUI) and presenting data to the user, ensures that users can visually interact with the different data structures (list, queue, and stack) in an intuitive manner. This package includes `AbstractViewFactory`, `SListView`, `QueueView` and `StackView`.

- `AbstractViewFactory` is the abstract class that provides a template or a set of common functionalities for creating views. This is the superclass of `SListView`, `QueueView` and `StackView` classes.
- `SListView` class represents the view for the singly linked list data structure, inherits from `AbstractViewFactory` and includes functionalities specific to displaying and interacting with lists.
- `QueueView` class represents the view for the queue data structure, extends `AbstractViewFactory` and includes functionalities specific to displaying and interacting with queue.
- `StackView` class represents the view for the stack data structure, extends `AbstractViewFactory` and includes functionalities specific to displaying and interacting with stack.

3.4 Controller package

The **Controller** package is responsible for processing user inputs, interacting with the **Model** to perform operations, and updating the **View** to reflect any changes. This package includes `AbstractControllerFactory`, `MainMenuController`, `ChatBoxController`, `SListController`, `QueueController`, `StackController`.

- `AbstractControllerFactory` is an abstract class that provides a template or a set of common functionalities for creating controllers. This is the superclass of `SListController`, `QueueController` and `StackController`.
- `MainMenuController` class handles the logic and interactions related to the main menu of the application. It processes user inputs from the main menu and coordinates with the **Model** and **View** to update the interface accordingly. It is not directly related to `AbstractControllerFactory` but interacts with other controllers and the main application flow.
- `ChatBoxController` class manages the interactions and logic for the chatbot feature of the application, handles user queries, processes them using AI, and provides responses to the user. Like `MainMenuController` class, it is not directly related to `AbstractControllerFactory` but interacts with the model to fetch data from users and provide responses.
- `SListController` class manages the interactions and logic for the list data structure. It inherits from `AbstractControllerFactory` class.
- `QueueController` class manages the interactions and logic for the queue data structure. It inherits from `AbstractControllerFactory` class.
- `StackController` class manages the interactions and logic for the stack data structure, inherits from `AbstractControllerFactory` class.

4 Detailed class diagram

In this section, our group will show the detailed class diagram, including each class in each package within the diagram.

4.1 Model package

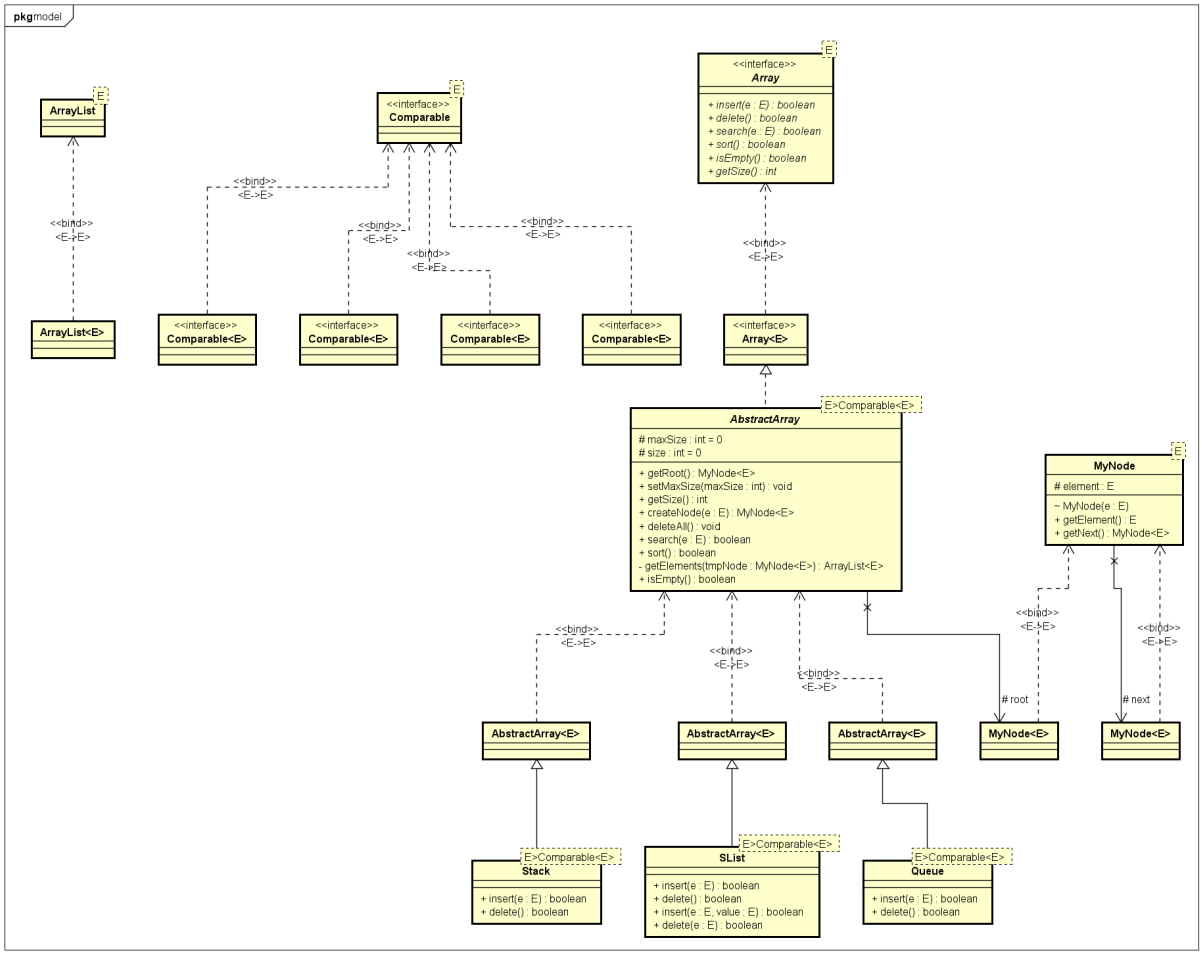


Figure 8: Model Package In Class Diagram

4.2 View package

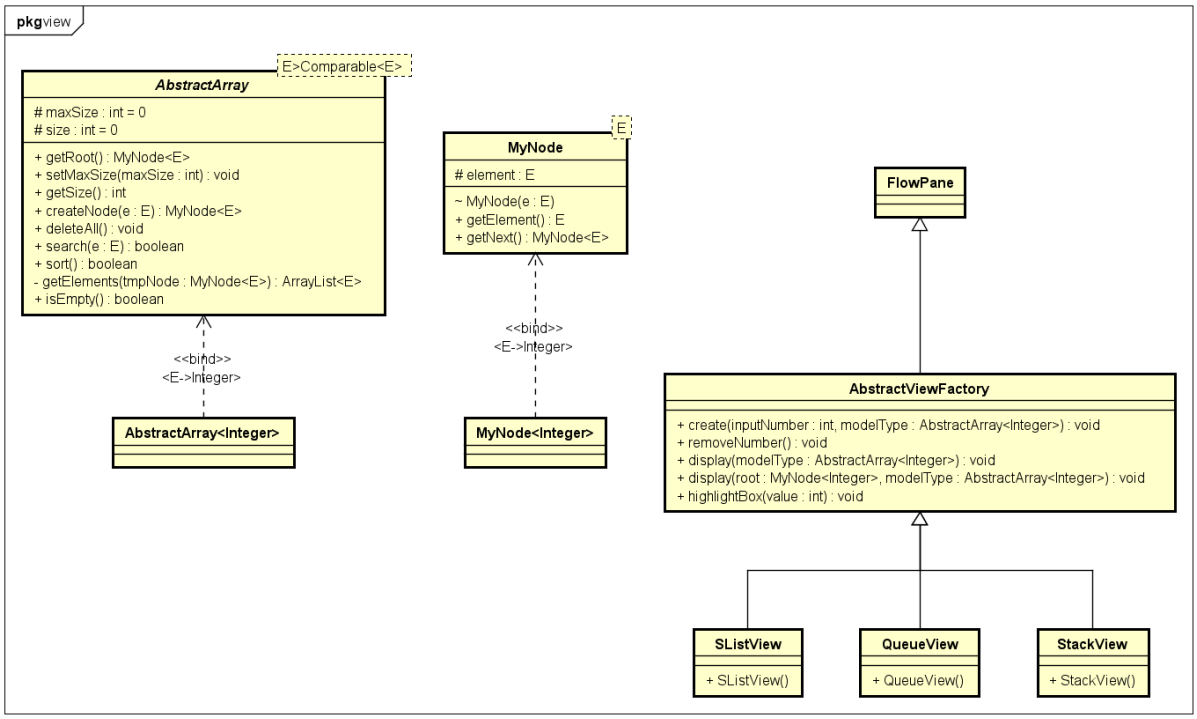


Figure 9: View Package In Class Diagram

4.3 Controller package

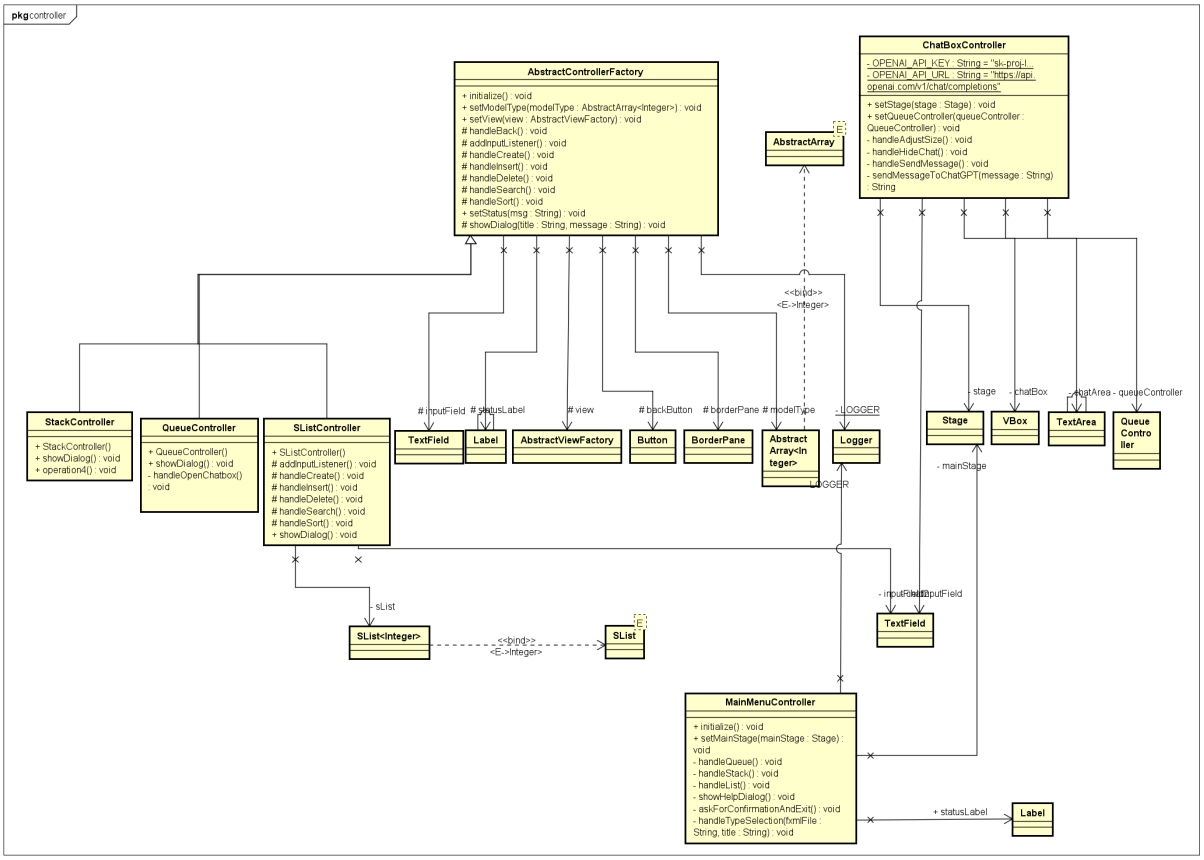


Figure 10: Controller Package In Class Diagram

5 Demo & Results

Our team has created an Java application that brings essential data structures like stack, queue and list. The app boasts a clean and straightforward interface, making it easy for users to interact with and understand. It's designed to be both user-friendly and intuitive, ensuring that anyone can navigate through its features without any confusion. To further boost the user experience, we've integrated an AI-powered chatbot. This chatbot is using an API from OpenAI based on ChatGPT. It serves as a helpful guide, offering real-time support and explanations about the various data structure operations.

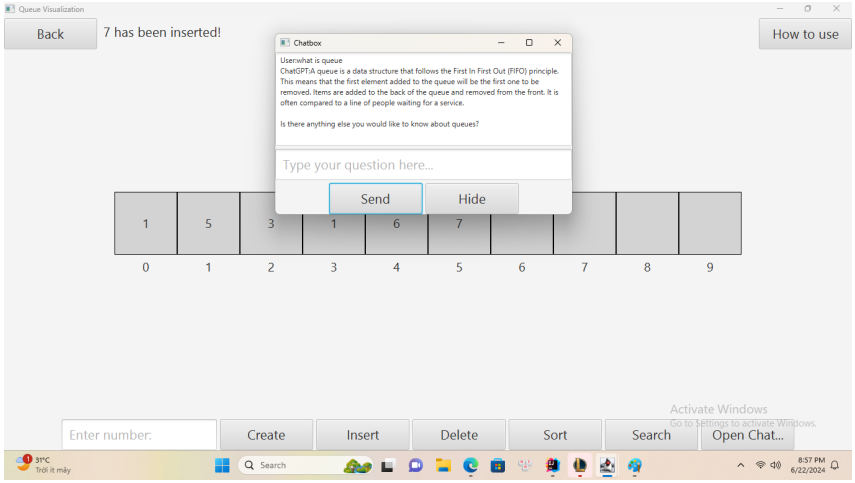


Figure 11: Queue visualizer with ChatGpt assistant

6 Contributions

In this section, we will highlight each group member's contributions to this project, providing an overview of the works they have contributed to.

- **Tran Cao Phong - 20226061 (Leader)**
  - Queue controller, app, view, model implementation (100%)
  - Chatbot implementation (100%)
  - GUI designing (100%)
  - Design use case diagram (10%)
  - Design general class diagram (10%)
  - Design detailed class diagram (10%)
- **Dang Van Nhan - 20225990**
  - Stack controller, app, view, model implementation (100%)
  - App testing & Checking for errors (100%)
  - Report writing (100%)
  - Design use case diagram (10%)
  - Design general class diagram (10%)
  - Design detailed class diagram (10%)
- **Bui Ha My - 20225987**
  - List controller, app, view, model implementation (50%)
  - Design use case diagram (10%)
  - Design general class diagram (70%)
  - Design detailed class diagram (70%)

- Slide preparing (50%)

- **Tran Kim Cuong - 20226017**
  - List controller, app, view, model implementation (50%)
  - Design use case diagram (70%)
  - Design general class diagram (10%)
  - Design detailed class diagram (10%)
  - Slide preparing (50%)

References

[1] GeeksforGeeks. *List Data Structure*. https://www.geeksforgeeks.org/list-interface-java-examples/.

[2] GeeksforGeeks. *MVC Framework Introduction*. https://www.geeksforgeeks.org/mvc-framework-introduction/.

[3] GeeksforGeeks. *Queue Data Structure*. https://www.geeksforgeeks.org/queue-data-structure/.

[4] GeeksforGeeks. *Stack Data Structure*. https://www.geeksforgeeks.org/stack-data-structure/.