Student Name : Ahmad Marwan AlMaghaireh  
ID : 23110205

Data Structures & Algorithms

Contents

[1. Provide a design specification using class diagram for the used data structures, explaining their valid operations. The design should consider a set of different ADTs for each single feature. Naming; List,Stack, Queue, Trees(BST), and Graph. 1](#_Toc200911035)

[2. Implement the provided solution using JAVA programming language, provided with a user menu,where user can select any option to execute. In addition to provide error handling cases where needed.Explain with use cases how your implementation solves the given problem, and justify your decision of ADTs and demonestrate how the system meets basic user needs. Evaluate three benifits of using the selected independed ADTs 13](#_Toc200911036)

[3. Critical evaluate the complexity of each implemented algorithm and ADT in the proposed solution. 60](#_Toc200911037)

[4. For feature #3, illustarte step by step in details the implemented algorithms. 64](#_Toc200911038)

[5. In reference to your solution, explain how information hiding and encapuslation helped you when 70](#_Toc200911039)

[using the implemented ADTs. Discuss with justification the view that ADTs are a basis for OOP. 70](#_Toc200911040)

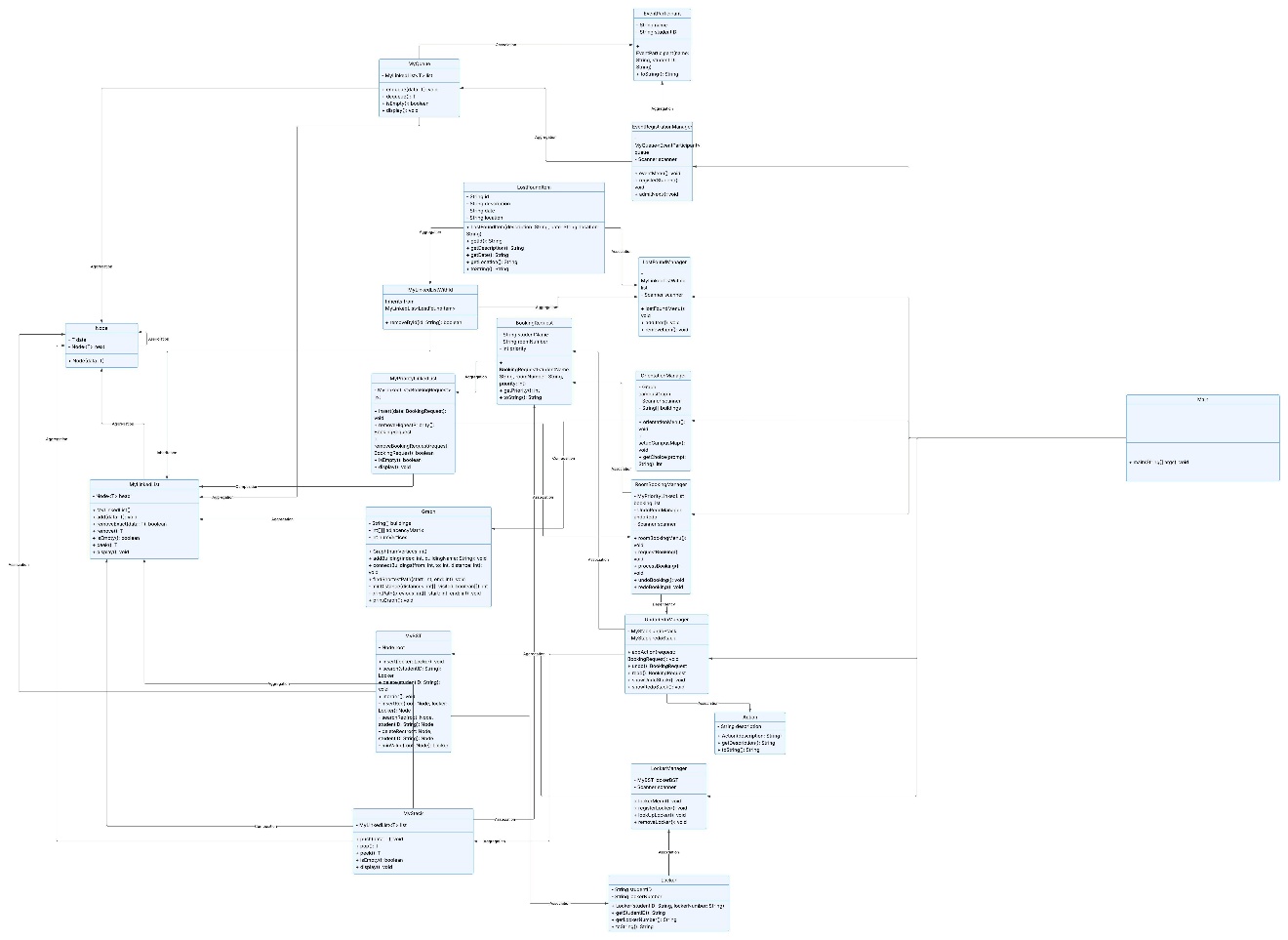
[6. For each implemented feature using a selected ADT, interpret the trade-off with it's alternatives in 72](#_Toc200911041)

[terms of time and space complexities for all valid operations, support your answer with examples. For 72](#_Toc200911042)

[example, if LinkedList ADT were chosen to implement feature #1, compare it with other possible 72](#_Toc200911043)

[alternative ADTs to impleement same feature such as ArrayList in terms of insert, remove, ..etc. 72](#_Toc200911044)

# 1. Provide a design specification using class diagram for the used data structures, explaining their valid operations. The design should consider a set of different ADTs for each single feature. Naming; List,Stack, Queue, Trees(BST), and Graph.



Important note :

1. Follow this link :

<https://lucid.app/lucidchart/bd99b860-ce0e-448c-a48a-dbd17136b53e/edit?viewport_loc=7420%2C-2148%2C3445%2C1519%2C0_0&invitationId=inv_e4e35d46-9d55-4a34-a3ee-14296e13bea7>

1. if he asks you to sign up use temp mail and sign up to see the diagram

temp mail : <https://temp-mail.org>

**1. Main Class**

* **Purpose**: Acts as the entry point for the Campus Management System. It manages the main menu and interacts with the feature managers like LostFoundManager, RoomBookingManager, EventRegistrationManager, etc.
* **Attributes**: None
* **Operations**:
  + main(String[] args) — Entry point for the system, where the user interacts with the system through the console menu.
* **Relations**:
  + It calls lostFoundManager, roomBookingManager, orientationManager, eventManager, and lockerManager to trigger their respective functionalities.

**2. Graph Class**

* **Purpose**: Represents the campus layout as a graph using an adjacency matrix to store connections between buildings.
* **Attributes**:
  + buildings: An array holding the names of the buildings.
  + adjacencyMatrix: A 2D array representing the connections and distances between buildings.
  + numVertices: The number of buildings.
* **Operations**:
  + addBuilding(int index, String buildingName): Adds a building to the campus graph.
  + connectBuildings(int from, int to, int distance): Connects two buildings with a distance.
  + findShortestPath(int start, int end): Uses Dijkstra’s algorithm to find the shortest path between two buildings.
  + printGraph(): Displays the adjacency matrix representation of the graph.
* **Relations**:
  + **Has a** connection with MyLinkedList indirectly, since it uses the Graph to manage the layout.

**3. MyBST Class**

* **Purpose**: Implements a **Binary Search Tree (BST)** to manage Locker objects by student ID.
* **Attributes**:
  + root: The root node of the BST.
* **Operations**:
  + insert(Locker locker): Adds a locker to the BST.
  + search(String studentID): Searches for a locker by student ID.
  + delete(String studentID): Deletes a locker based on student ID.
  + inorder(): Performs in-order traversal and displays all locker information.
* **Relations**:
  + **Has a** relation with Locker, as Locker objects are stored in the BST.

**4. MyLinkedList Class**

* **Purpose**: Implements a singly linked list, which is used by various other classes like MyQueue, MyStack, and MyPriorityLinkedList.
* **Attributes**:
  + head: The first node of the list.
* **Operations**:
  + add(T data): Adds a new node with the data.
  + removeExact(T data): Removes a specific node from the list.
  + remove(): Removes the first node in the list.
  + isEmpty(): Checks if the list is empty.
  + peek(): Returns the data of the first node.
  + display(): Displays all the nodes in the list.
* **Relations**:
  + **Has a** relation with Node, as nodes make up the linked list.

**5. MyLinkedListWithId Class**

* **Purpose**: Extends MyLinkedList and adds functionality to remove a LostFoundItem by ID.
* **Attributes**: Inherits from MyLinkedList<LostFoundItem>.
* **Operations**:
  + removeById(String id): Removes an item from the list by its unique ID.
* **Relations**:
  + **Uses** MyLinkedList as its base class and **manages** LostFoundItem objects.

**6. MyPriorityLinkedList Class**

* **Purpose**: Implements a priority queue using a linked list, where elements are inserted based on priority.
* **Attributes**:
  + list: A MyLinkedList<BookingRequest> that holds the requests in order of priority.
* **Operations**:
  + insert(BookingRequest data): Inserts a booking request in priority order.
  + removeHighestPriority(): Removes and returns the highest priority booking.
  + removeBookingRequest(BookingRequest request): Removes the specific booking request.
  + isEmpty(): Checks if the priority list is empty.
  + display(): Displays all booking requests.
* **Relations**:
  + **Uses** MyLinkedList for internal data storage of BookingRequest objects.
  + **Has a** relation with BookingRequest.

**7. MyQueue Class**

* **Purpose**: Implements a queue data structure using MyLinkedList.
* **Attributes**:
  + list: A MyLinkedList<T> used to store queue elements.
* **Operations**:
  + enqueue(T data): Adds an item to the end of the queue.
  + dequeue(): Removes and returns the first item from the queue.
  + isEmpty(): Checks if the queue is empty.
  + display(): Displays all elements in the queue.
* **Relations**:
  + **Uses** MyLinkedList to manage the data structure.

**8. MyStack Class**

* **Purpose**: Implements a stack data structure using MyLinkedList.
* **Attributes**:
  + list: A MyLinkedList<T> used to store stack elements.
* **Operations**:
  + push(T data): Pushes an item onto the stack.
  + pop(): Removes and returns the top item from the stack.
  + peek(): Returns the top item without removing it.
  + isEmpty(): Checks if the stack is empty.
  + display(): Displays all stack elements.
* **Relations**:
  + **Uses** MyLinkedList to manage the data structure.

**9. Node Class**

* **Purpose**: Represents a node in a linked list, stack, or queue.
* **Attributes**:
  + data: The data stored in the node.
  + next: A reference to the next node in the list.
* **Operations**:
  + Constructor to initialize the node with data.
* **Relations**:
  + **Used by** MyLinkedList, MyStack, MyQueue, MyPriorityLinkedList to hold the elements.

**Feature Classes**

**10. EventRegistrationManager Class**

* **Purpose**: Manages event registration and queueing of event participants.
* **Attributes**:
  + queue: A MyQueue<EventParticipant> to manage event participants.
  + scanner: A scanner for user input.
* **Operations**:
  + eventMenu(): Displays the event registration menu.
  + registerStudent(): Registers a student for the event.
  + admitNext(): Admits the next student from the queue.
* **Relations**:
  + **Uses** MyQueue to manage participants.
  + **Has a** relation with EventParticipant.

**11. LockerManager Class**

* **Purpose**: Manages locker assignments using a binary search tree.
* **Attributes**:
  + lockerBST: A MyBST that stores locker assignments.
  + scanner: A scanner for user input.
* **Operations**:
  + lockerMenu(): Displays the locker management menu.
  + registerLocker(): Registers a locker for a student.
  + lookUpLocker(): Looks up a locker by student ID.
  + removeLocker(): Removes a student's locker.
* **Relations**:
  + **Uses** MyBST to store locker assignments.
  + **Has a** relation with Locker.

**12. LostFoundManager Class**

* **Purpose**: Manages lost and found items, including adding and removing items from the list.
* **Attributes**:
  + list: A MyLinkedListWithId that stores lost and found items.
  + scanner: A scanner for user input.
* **Operations**:
  + lostFoundMenu(): Displays the lost and found menu.
  + addItem(): Adds a lost or found item.
  + removeItem(): Removes an item by its unique ID.
* **Relations**:
  + **Uses** MyLinkedListWithId to store lost and found items.
  + **Has a** relation with LostFoundItem.

**13. OrientationManager Class**

* **Purpose**: Provides campus orientation and calculates the shortest path between buildings.
* **Attributes**:
  + campusGraph: A Graph representing the campus layout.
  + scanner: A scanner for user input.
  + buildings: An array holding the names of the buildings.
* **Operations**:
  + orientationMenu(): Displays the orientation menu.
  + setupCampusMap(): Sets up the map with buildings and connections.
  + getChoice(): Prompts the user to make a choice.
* **Relations**:
  + **Uses** Graph to manage the campus building layout.
  + **Has a** relation with Graph.

**14. RoomBookingManager Class**

* **Purpose**: Manages room booking requests and integrates undo/redo functionality.
* **Attributes**:
  + bookingList: A MyPriorityLinkedList that stores booking requests.
  + undoRedo: A UndoRedoManager for managing undo/redo actions.
  + scanner: A scanner for user input.
* **Operations**:
  + roomBookingMenu(): Displays the room booking menu.
  + requestBooking(): Allows users to request a room booking.
  + processBooking(): Processes the highest priority booking.
  + undoBooking(): Undoes the last booking.
  + redoBooking(): Redoes the last undone booking.
* **Relations**:
  + **Uses** MyPriorityLinkedList to store booking requests.
  + **Has a** relation with BookingRequest.
  + **Depends on** UndoRedoManager for undo/redo functionality.

**15. UndoRedoManager Class**

* **Purpose**: Manages undo and redo operations for room bookings.
* **Attributes**:
  + undoStack: A MyStack of undo actions.
  + redoStack: A MyStack of redo actions.
* **Operations**:
  + addAction(): Adds an action to the undo stack.
  + undo(): Undoes the last action.
  + redo(): Redoes the last undone action.
  + showUndoStack(): Displays the undo stack.
  + showRedoStack(): Displays the redo stack.
* **Relations**:
  + **Uses** MyStack to manage undo and redo actions.
  + **Has a** relation with BookingRequest.

**16. Action Class**

* **Purpose**: Represents an action (such as a booking request) that can be undone or redone.
* **Attributes**:
  + description: A string describing the action.
* **Operations**:
  + getDescription(): Retrieves the description of the action.
  + toString(): Returns a string representation of the action.
* **Relations**:
  + **Used by** UndoRedoManager to store the actions.

**17. BookingRequest Class**

* **Purpose**: Represents a booking request for a room.
* **Attributes**:
  + studentName: The name of the student making the booking.
  + roomNumber: The room number being booked.
  + priority: The priority of the booking request.
* **Operations**:
  + getPriority(): Returns the priority of the request.
  + toString(): Returns a string representation of the booking request.
* **Relations**:
  + **Used by** RoomBookingManager and MyPriorityLinkedList.

**18. EventParticipant Class**

* **Purpose**: Represents a participant in an event.
* **Attributes**:
  + name: The participant's name.
  + studentID: The participant's student ID.
* **Operations**:
  + toString(): Returns a string representation of the participant.
* **Relations**:
  + **Used by** EventRegistrationManager.

**19. Locker Class**

* **Purpose**: Represents a locker assigned to a student.
* **Attributes**:

studentID: The student's ID.

lockerNumber: The locker number.

* **Operations**:

getStudentID(): Returns the student's ID.

getLockerNumber(): Returns the locker number.

toString(): Returns a string representation of the locker.

* **Relations**:

**Used by** MyBST and LockerManager.

**20. LostFoundItem Class**

* **Purpose**: Represents a lost or found item.
* **Attributes**:

id: The unique ID for the item.

description: A description of the item.

date: The date the item was lost or found.

location: The location of the item.

* **Operations**:

getId(): Returns the unique ID of the item.

getDescription(): Returns the description of the item.

getDate(): Returns the date the item was lost or found.

getLocation(): Returns the location of the item.

toString(): Returns a string representation of the item.

* **Relations**:

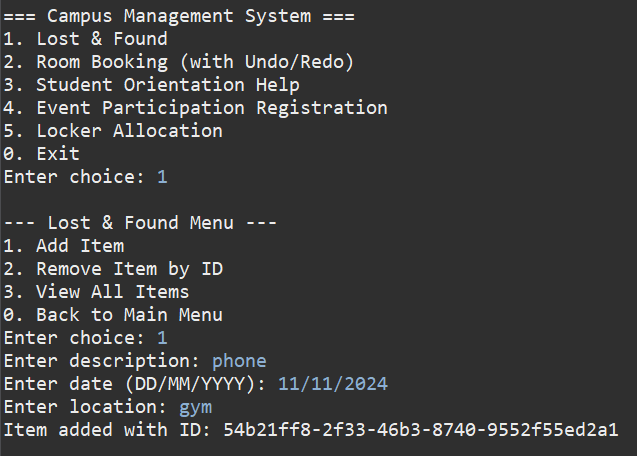
**Used by** MyLinkedListWithId and LostFoundManager.

# 2. Implement the provided solution using JAVA programming language, provided with a user menu,where user can select any option to execute. In addition to provide error handling cases where needed.Explain with use cases how your implementation solves the given problem, and justify your decision of ADTs and demonestrate how the system meets basic user needs. Evaluate three benifits of using the selected independed ADTs

**Use Cases, Error Handling, and ADT Explanation for Campus Management System**

**Feature 1: Track Lost/Found Record**

* **Purpose**: Track lost and found items, storing information like description, date, and location.
* **ADT Used**: MyLinkedListWithId
* **Explanation**:
  + LostFoundManager interacts with MyLinkedListWithId to add, remove, and display lost/found items.
  + Note that MyLinkedListWithId is an extended for MyLinkedList but to keep the concept of ADT and separation of duties I used MyLinkedListWithId to add a method removeById without affecting the parent MyLinkedList which is used by other ADTs across the project
  + Items are represented by LostFoundItem objects that store descriptions, dates, and locations.
* **Use Case**:
  + **Add Item**: A user adds a lost item with description, date, and location. The LostFoundManager creates a new LostFoundItem and adds it to the MyLinkedListWithId using add() method.



* + **Remove Item**: The user can remove an item by its ID using removeById() method in MyLinkedListWithId.

A screenshot of a computer

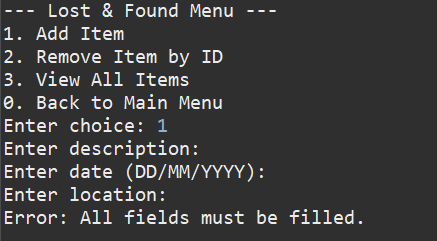
AI-generated content may be incorrect.

* + **View Items**: A user can view all lost/found items using the display() method of MyLinkedListWithId.

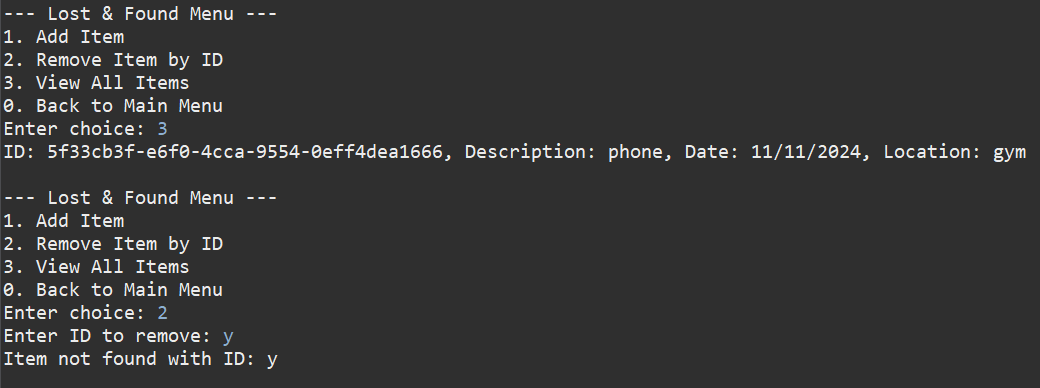
A screen shot of a computer

AI-generated content may be incorrect.

* **Error Handling**:
  + If the fields (description, date, location) are empty, an error message is displayed, e.g., Error: All fields must be filled.

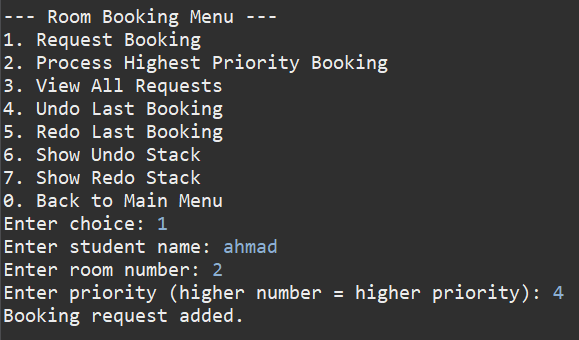


* + If an invalid ID is given to remove an item, the message Item not found with ID: {id} is shown.

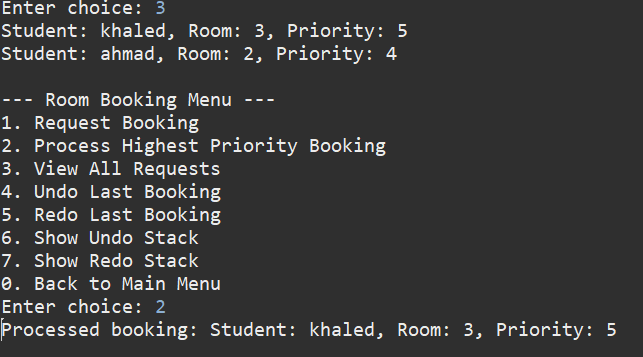


**Feature 2: Room Booking**

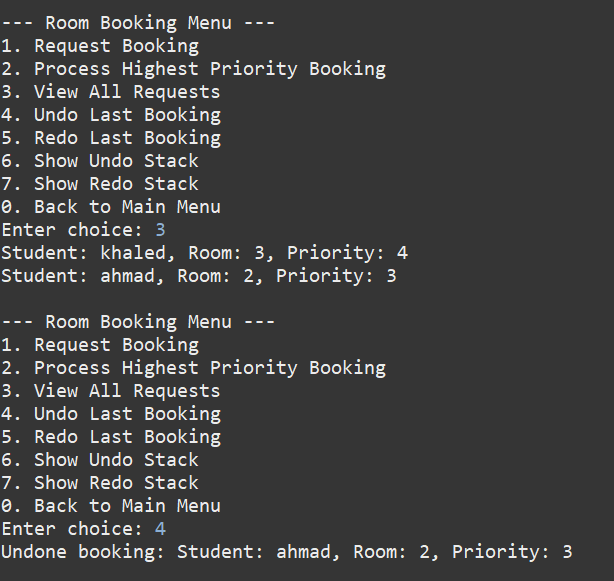
* **Purpose**: Students can request bookings for private study rooms based on priority.
* **ADT Used**: MyPriorityLinkedList
* **Explanation**:
  + RoomBookingManager uses MyPriorityLinkedList to manage booking requests. Requests are stored and ordered based on priority.
  + Each request is a BookingRequest object that includes a student's name, room number, and priority level.
* **Use Case**:
  + **Request Booking**: The user submits a room booking with priority. The RoomBookingManager inserts the BookingRequest into MyPriorityLinkedList based on priority.



* + **Process Booking**: The system processes the highest priority booking using removeHighestPriority(), removing it from the list.



* + **Undo/Redo**: The system supports undoing and redoing booking requests through the UndoRedoManager.



A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

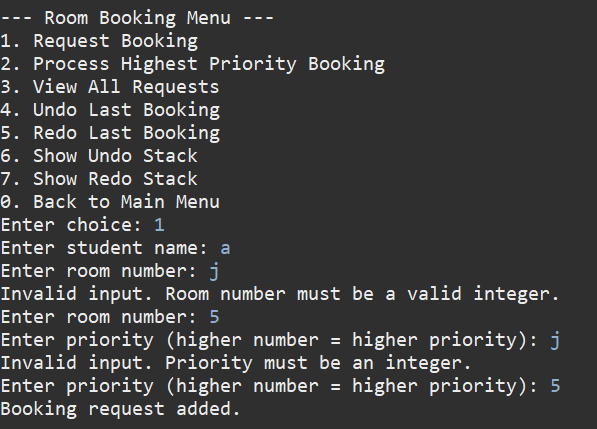
AI-generated content may be incorrect.

* **Error Handling**:
  + If the user tries to process a booking when the list is empty, it displays the message No bookings to process.

A screenshot of a computer program

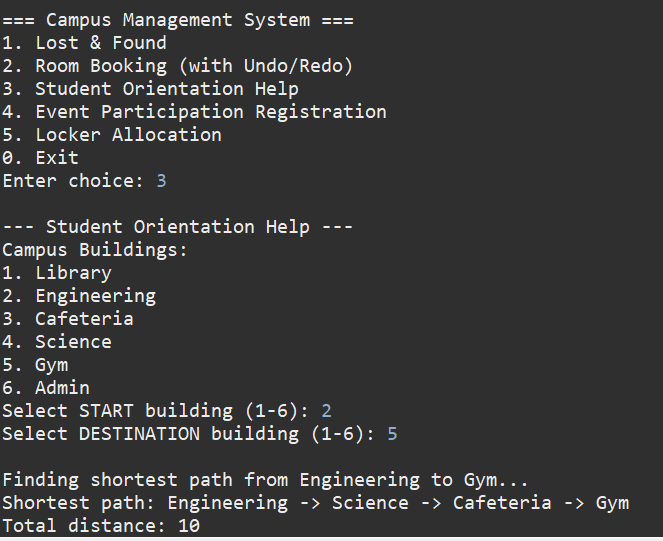
AI-generated content may be incorrect.

* + If invalid input is entered during the booking process, such as a non-integer value for priority, the system prompts Invalid input. Please enter a valid number.



**Feature 3: Student Orientation Help**

* **Purpose**: Guide new students to navigate between buildings and find the shortest path on campus.
* **ADT Used**: Graph
* **Explanation**:
  + OrientationManager uses Graph to represent campus buildings and connections.
  + Buildings are stored in an adjacency matrix, and the shortest path is calculated using Dijkstra's algorithm.
* **Use Case**:
  + **Select Start/End Building**: The user selects the starting and destination buildings, and findShortestPath() is called to calculate the shortest route.



A screen shot of a computer program

AI-generated content may be incorrect.

* + **Find Shortest Path**: The Graph calculates the shortest path using Dijkstra's algorithm, printing the path and the total distance.

A computer screen shot of a program

AI-generated content may be incorrect.

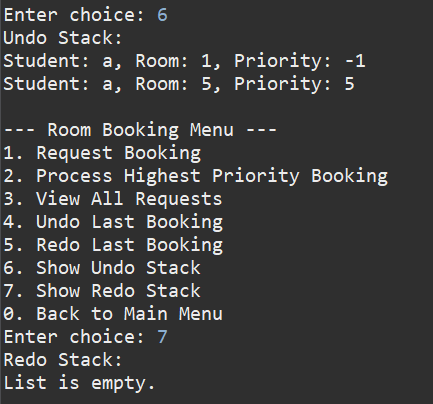
* **Error Handling**:
  + If the user selects an invalid building index, the system displays Invalid building selections.

A screen shot of a computer program

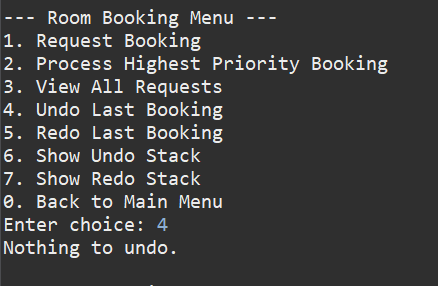
AI-generated content may be incorrect.

**Feature 4: Undo/Redo**

* **Purpose**: Allow users to undo or redo their actions, like room bookings or locker assignments.
* **ADT Used**: MyStack
* **Explanation**:
  + UndoRedoManager uses two stacks (undoStack, redoStack) to store and manage actions.
  + Whenever a user performs an action, it is added to the undoStack, and the redoStack is cleared.
* **Use Case**:
  + **Undo Action**: A user can undo the last action (e.g., room booking) using undo().

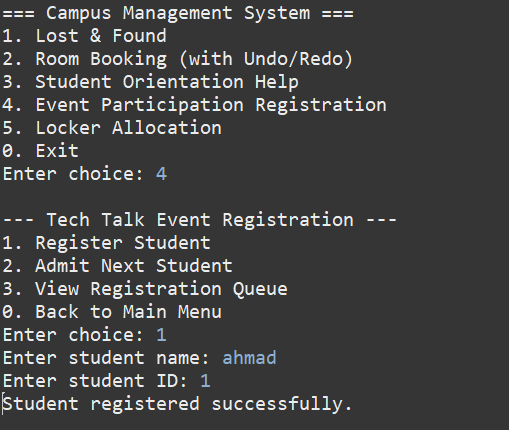


* + **Redo Action**: A user can redo an undone action using redo().
* **Error Handling**:
  + If there are no actions to undo or redo, it displays Nothing to undo. or Nothing to redo..

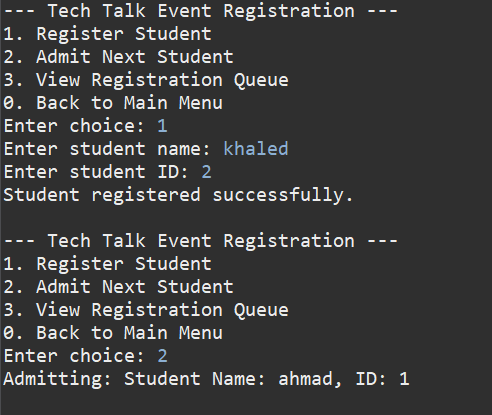


**Feature 5: Event Participation Registration**

* **Purpose**: Register students for events and track registrations in the order they were received.
* **ADT Used**: MyQueue
* **Explanation**:
  + EventRegistrationManager uses MyQueue to manage student registrations. The queue follows a First-In-First-Out (FIFO) order for participants.
  + Each participant is represented by an EventParticipant object.
* **Use Case**:
  + **Register Student**: A user registers a student for an event, and the student is added to the queue.



* + **Admit Next Student**: The system admits the next student in the queue and removes them from the list.



* + **View Registrations**: The system displays the list of registered participants.

A screen shot of a computer

AI-generated content may be incorrect.

* **Error Handling**:
  + If the queue is empty when attempting to admit a student, it displays No students waiting.

A screen shot of a computer

AI-generated content may be incorrect.

**Feature 6: Student Locker Allocation**

* **Purpose**: Assign lockers based on student ID and allow quick lookup, removal, or registration as students graduate or withdraw.
* **ADT Used**: MyBST
* **Explanation**:
  + LockerManager uses MyBST (Binary Search Tree) to manage locker assignments. Lockers are identified by student IDs, and the system can perform fast search, insert, and delete operations on the tree.
  + Each locker is represented by a Locker object.
* **Use Case**:
  + **Register Locker**: A user can register a locker by entering the student ID and locker number. The system adds the Locker to the MyBST.

A screenshot of a computer program

AI-generated content may be incorrect.

* + **Look Up Locker**: A user can search for a locker by student ID using the search() method in MyBST.

A screen shot of a computer

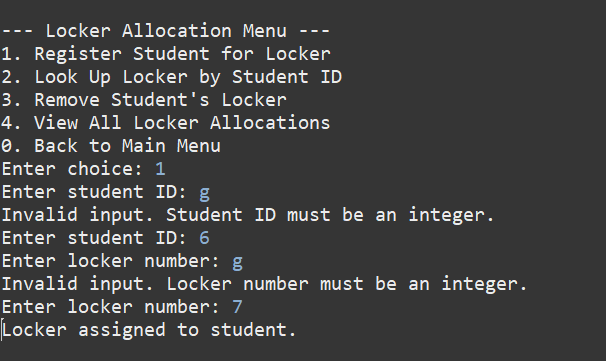
AI-generated content may be incorrect.

* + **Remove Locker**: The system allows removal of lockers when students graduate or withdraw by using the delete() method.

A screenshot of a computer program

AI-generated content may be incorrect.

* **Error Handling**:
  + If the student ID is invalid, the system shows Error: Invalid student ID.



* + If the locker is not found during lookup, it displays Locker not found for this student..

A screen shot of a computer

AI-generated content may be incorrect.

**Summary of ADT Decisions**

1. **Graph** (used in Student Orientation Help):

**Why**: A graph is ideal for representing the campus buildings and their connections. The adjacency matrix efficiently handles the bidirectional nature of the connections between buildings.

**Operations**: Add buildings, connect buildings, find the shortest path using Dijkstra’s algorithm.

1. **MyBST** (used in Locker Allocation):

**Why**: A binary search tree (BST) is well-suited for storing locker assignments by student ID. The tree allows for efficient search, insertion, and deletion operations based on student ID.

**Operations**: Insert, search, delete, in-order traversal.

1. **MyLinkedList** (used in Event Registration and Lost/Found Items):

**Why**: A linked list is perfect for managing items or participants in a FIFO (for event registration) or non-FIFO (for lost/found items) manner.

**Operations**: Add, remove, peek, display.

1. **MyPriorityLinkedList** (used in Room Booking):

**Why**: A priority linked list ensures booking requests are stored and processed based on priority, which is required for handling room booking requests.

**Operations**: Insert based on priority, remove highest priority, display.

1. **MyQueue** (used in Event Registration):

**Why**: A queue is ideal for managing event participant registration in the order they are received (FIFO).

**Operations**: Enqueue, dequeue, display.

1. **MyStack** (used in Undo/Redo):

**Why**: A stack is ideal for implementing undo and redo operations. Actions can be pushed to the stack, and the last action can be popped for undo or redo.

**Operations**: Push, pop, peek, display.

**Explanation for every feature :**  
  
**Understanding Feature 1: Track Lost/Found Record**

**Big Picture: Relation Between ADTs, Feature, and Models**

In **Feature 1**, we are dealing with the **Lost and Found** functionality of the Campus Management System. The objective is to track lost and found items, store them with details such as description, date, and location.

1. **Feature Implementation:**
   * **LostFoundManager** manages the feature, providing methods to add, remove, and display lost/found items.
   * It uses an **ADT (MyLinkedListWithId)** to store lost and found items in a linked list structure.
   * Each **LostFoundItem** (model) represents an individual lost or found item with its details (description, date, location).
2. **ADTs Involved:**
   * **MyLinkedList**: This generic linked list is used as the underlying structure for **MyLinkedListWithId**.
   * **MyLinkedListWithId**: Inherits from **MyLinkedList** and adds functionality for removing items based on their **ID**. This ensures that the list of lost and found items can be efficiently managed and updated.
3. **Model:**
   * **LostFoundItem**: Represents a lost/found item, with attributes like description, date, location, and a unique ID generated using UUID. The model ensures that every item has a distinct identifier, which is crucial for item retrieval and removal.

**Feature Flow**

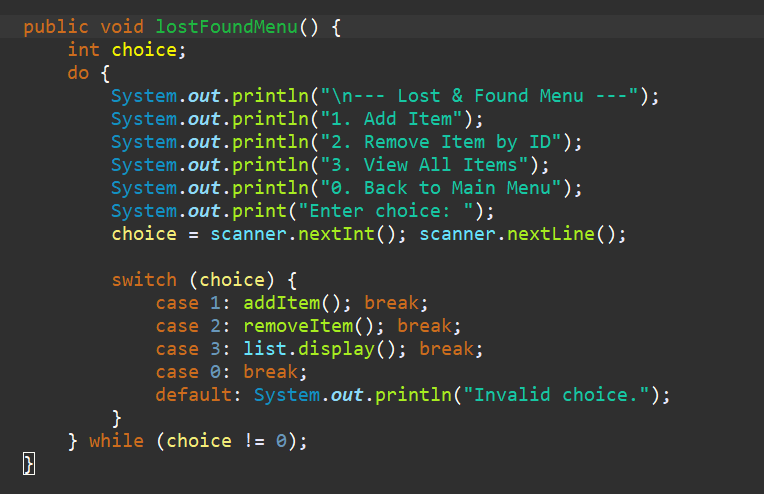
1. **Add Lost/Found Item**:
   * The user provides details such as description, date, and location.
   * **LostFoundManager** creates a new **LostFoundItem** and adds it to **MyLinkedListWithId** using the add() method.
   * **MyLinkedListWithId** stores these items in a linked list.
   * **LostFoundItem** is added with a **unique ID** generated automatically.
2. **Remove Item by ID**:
   * When a user wants to remove an item, they provide the ID.
   * **LostFoundManager** calls the removeById() method in **MyLinkedListWithId**, which searches for the item by ID and removes it from the list.
3. **Display Lost/Found Items**:
   * **LostFoundManager** can display all items stored in the list using the display() method in **MyLinkedList**.

**ADTs Tracing for LostFoundManager**

Let's trace the ADTs involved, their operations, and the flow of data:

**1. LostFoundManager (Feature Implementation)**

* **Attributes**:
  + MyLinkedListWithId list: This is the core ADT used to store lost/found items.
  + Scanner scanner: To interact with the user for input.
* **Operations**:
  + lostFoundMenu(): Displays the main menu for the Lost & Found feature, allowing users to choose between adding, removing, or displaying items.



* + addItem(): Adds a lost/found item to the list.

A screen shot of a computer code

AI-generated content may be incorrect.

* + removeItem(): Removes an item by its unique ID from the list.

A computer screen shot of code

AI-generated content may be incorrect.

**2. MyLinkedListWithId (ADT used for storing lost/found items)**

* **Attributes**:
  + Inherits from **MyLinkedList** and stores **LostFoundItem** objects.



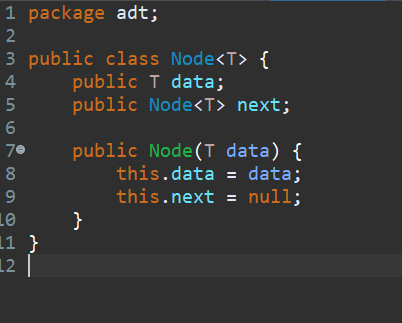
* **Operations**:
  + removeById(String id): Removes an item based on its unique ID by searching through the linked list. It ensures that the item is removed efficiently using object equality comparison.



* + The add(), removeExact(), isEmpty(), display(), and other methods come from the parent class **MyLinkedList**.

**3. MyLinkedList (Underlying ADT for MyLinkedListWithId)**

* **Attributes**:
  + Node<T> head: The starting point of the linked list.



* **Operations**:
  + add(T data): Adds a new node to the list.

A screen shot of a computer code

AI-generated content may be incorrect.

* + removeExact(T data): Removes a specific node based on its value (object equality).

A computer screen shot of code

AI-generated content may be incorrect.

* + isEmpty(): Checks if the list is empty.

A black screen with colorful text

AI-generated content may be incorrect.

* + display(): Displays the contents of the list by traversing it.

A screen shot of a computer code

AI-generated content may be incorrect.

**4. LostFoundItem (Model used in MyLinkedListWithId)**

* **Attributes**:
  + String id: A unique identifier for the lost/found item.
  + String description: Description of the item.
  + String date: Date when the item was lost/found.
  + String location: The location where the item was found or lost.

A screen shot of a computer code

AI-generated content may be incorrect.

* **Operations**:
  + getId(), getDescription(), getDate(), getLocation(): Methods to retrieve the details of the lost/found item.

A screen shot of a computer program

AI-generated content may be incorrect.

This feature provides a well-structured approach to managing lost and found items. The ADT MyLinkedListWithId efficiently stores items and ensures unique identification through **UUID**. The LostFoundManager orchestrates the user interaction, handling errors such as empty fields or invalid IDs with appropriate messages.

The combination of **MyLinkedList** and **MyLinkedListWithId** enables a flexible and efficient way to manage a dynamic collection of lost and found items. The **LostFoundItem** model encapsulates the data, ensuring each item has unique identification and relevant attributes (description, date, location)

**Feature 2: Room Booking**

**Big Picture: Relation Between ADTs, Feature, and Models**

**Feature 2** allows students to request room bookings for study sessions, process these bookings based on priority, and manage the ability to undo or redo the last booking. This feature primarily revolves around the **booking requests**, and the **priority** of those requests.

1. **Feature Implementation**:
   * **RoomBookingManager** handles the main functionality, including adding booking requests, processing them by priority, and managing undo/redo operations.
   * It uses **MyPriorityLinkedList** (an ADT) to manage the requests based on their priority. **BookingRequest** is the model that stores the request data, including student name, room number, and priority.
2. **ADTs Involved**:
   * **MyPriorityLinkedList**: This is a custom ADT based on a linked list that orders **BookingRequest** objects based on their priority. It ensures that the highest priority request is always processed first.
   * **MyLinkedList**: This ADT is used internally in **MyPriorityLinkedList** for handling the linked list structure. It provides the basic operations for adding, removing, and displaying nodes.
   * **MyStack** (via **UndoRedoManager**): This ADT is used to implement the undo/redo functionality, allowing users to reverse or reapply their last actions (e.g., booking requests).
3. **Model**:
   * **BookingRequest**: Represents a room booking request from a student, containing the student’s name, the room number, and the priority of the request.

**Feature Flow**

1. **Request Booking**:
   * The user (student) enters their name, the room number, and the priority of the request.
   * **RoomBookingManager** creates a new **BookingRequest** object and inserts it into the **MyPriorityLinkedList** using the insert() method. The requests are sorted in descending order of priority.
   * **MyPriorityLinkedList** uses an insertion sort to ensure that higher priority requests are always processed first.
2. **Process Booking**:
   * **RoomBookingManager** processes the highest priority booking by calling removeHighestPriority() on **MyPriorityLinkedList**. This removes the first item (highest priority) from the list.
3. **Undo Booking**:
   * **RoomBookingManager** allows users to undo the most recent booking request. The system uses **UndoRedoManager**'s undo() method to restore the previous state and remove the last booking request from the list.
4. **Redo Booking**:
   * **RoomBookingManager** also allows users to redo the most recently undone booking request. The system uses **UndoRedoManager**'s redo() method to add the previously undone request back into the list.
5. **Display All Requests**:
   * The user can view all the current room booking requests by calling the display() method in **MyPriorityLinkedList**, which internally calls the display() method from **MyLinkedList**.

**ADTs Tracing for RoomBookingManager**

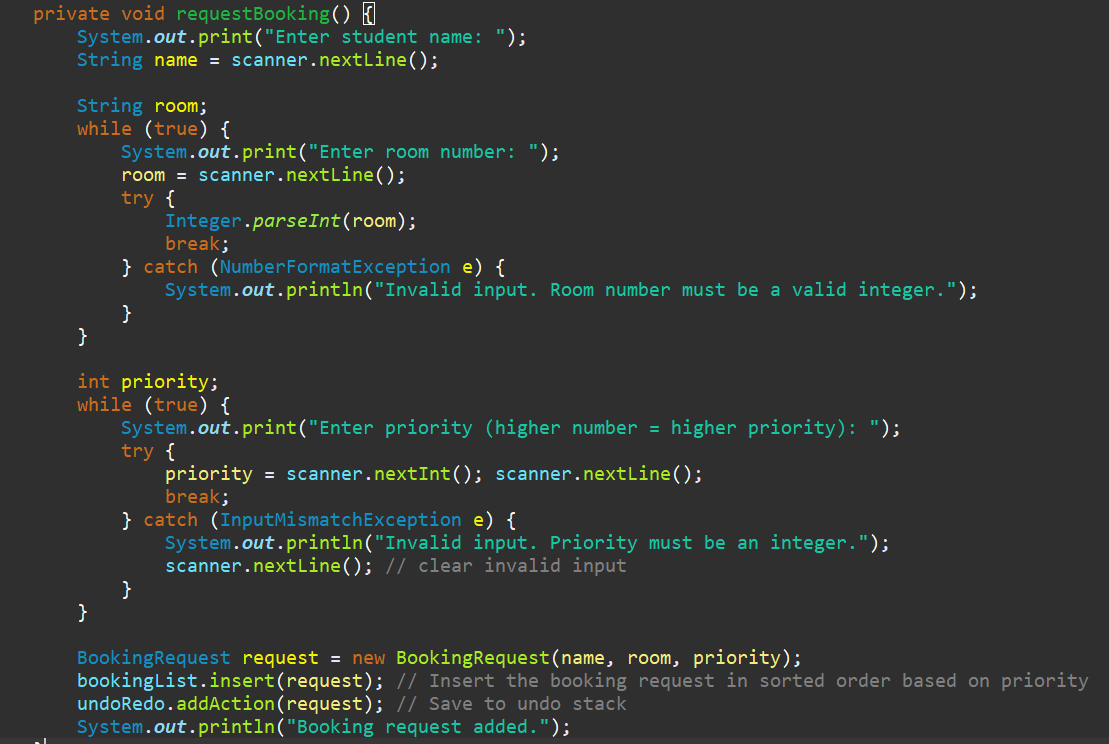
Let's trace the ADTs involved, their operations, and the flow of data:

**1. RoomBookingManager (Feature Implementation)**

* **Attributes**:
  + MyPriorityLinkedList bookingList: This ADT manages the room booking requests, sorted by priority.
  + UndoRedoManager undoRedo: This ADT manages the undo and redo functionality for booking actions.
  + Scanner scanner: Used for reading user input.
* **Operations**:
  + roomBookingMenu(): Displays the main menu for the Room Booking feature.



* + requestBooking(): Handles the process of collecting user input for a room booking request and adds it to the booking list.

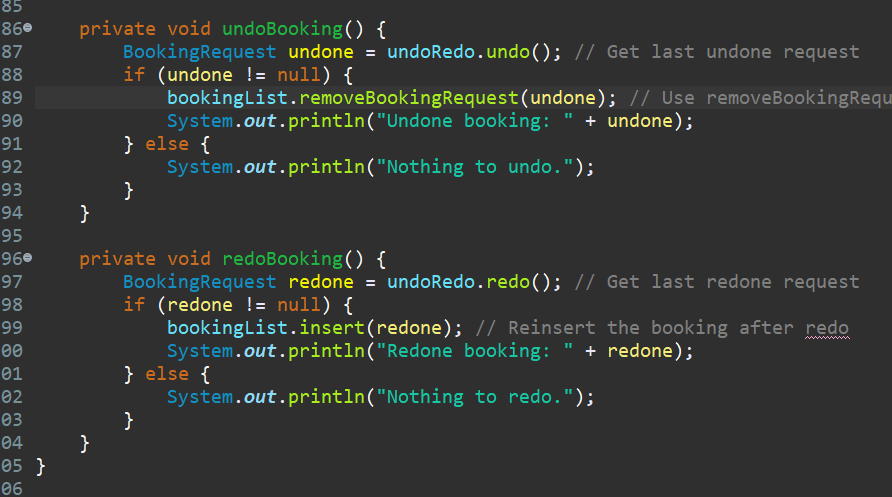


* + processBooking(): Removes and processes the highest priority booking request.

A screenshot of a computer program

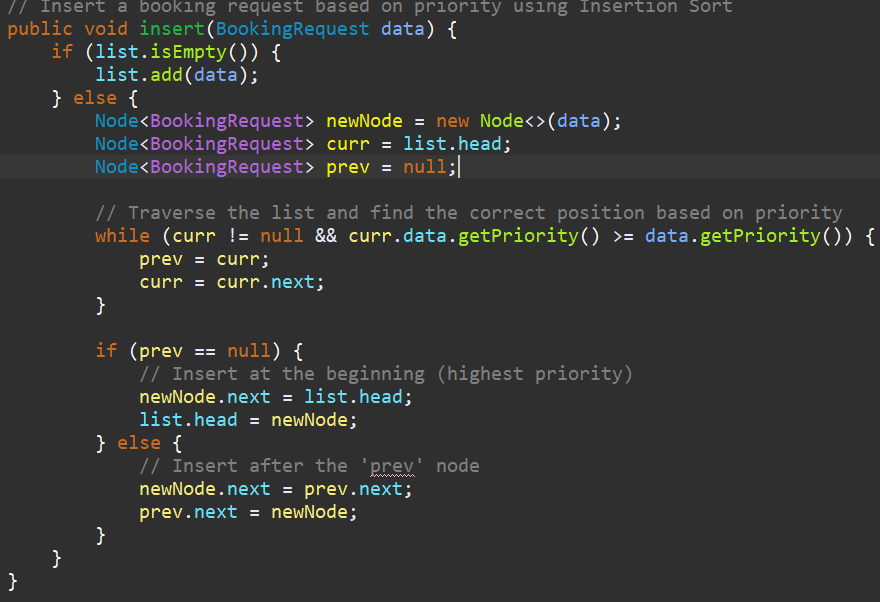
AI-generated content may be incorrect.

* + undoBooking(): Undoes the last room booking request.
  + redoBooking(): Redoes the most recent undone booking request.



**2. MyPriorityLinkedList (Priority Queue Implementation)**

* **Attributes**:
  + MyLinkedList<BookingRequest> list: Uses **MyLinkedList** as the underlying structure for storing **BookingRequest** objects.
* **Operations**:
  + insert(BookingRequest data): Inserts a booking request in the linked list, ensuring that the list is sorted by priority using an insertion sort.



* + removeHighestPriority(): Removes and returns the highest priority booking request (the first item in the list).

A screen shot of a computer code

AI-generated content may be incorrect.

* + removeBookingRequest(BookingRequest request): Removes a specific booking request from the list.

A screen shot of a computer program

AI-generated content may be incorrect.

* + isEmpty(): Checks if the list is empty.

A screen shot of a computer

AI-generated content may be incorrect.

* + display(): Displays all the booking requests in the list.

A screen shot of text

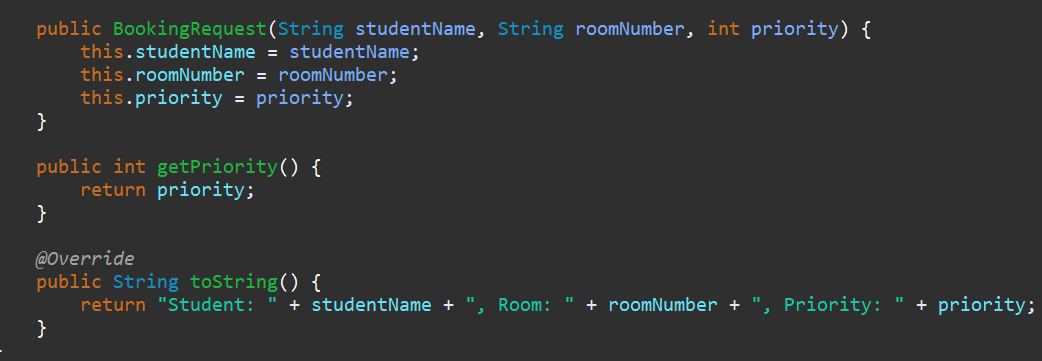
AI-generated content may be incorrect.

**3. MyLinkedList (Underlying ADT for MyPriorityLinkedList)**

* **Attributes**:
  + Node<T> head: The starting point of the linked list.
* **Operations**:
  + add(T data): Adds a new node to the list.
  + removeExact(T data): Removes a specific node based on its value (object equality).
  + isEmpty(): Checks if the list is empty.
  + display(): Displays the contents of the list by traversing it.

**4. BookingRequest (Model used in MyPriorityLinkedList)**

* **Attributes**:
  + String studentName: The name of the student requesting the room.
  + String roomNumber: The number of the room being requested.
  + int priority: The priority of the booking request, where higher values indicate higher priority.
* **Operations**:
  + getPriority(): Returns the priority of the booking request.
  + toString(): Returns a string representation of the booking request, including the student name, room number, and priority.



This feature provides a robust solution for managing room booking requests. The use of **MyPriorityLinkedList** ensures that the highest priority requests are always processed first, which is crucial for a booking system. **BookingRequest** encapsulates the data necessary for each booking, and **RoomBookingManager** efficiently manages the overall flow, including handling user input, processing requests, and managing undo/redo actions.

The **UndoRedoManager** allows the system to maintain the integrity of user actions by providing the ability to reverse or reapply the last booking request, ensuring flexibility in the system.

**Feature 3: Student Orientation Help**

**Big Picture: Relation Between ADTs, Feature, and Models**

**Feature 3** allows students to receive guidance on how to navigate between buildings on the campus and find the shortest path between two buildings using **Dijkstra’s Algorithm**.

* **Feature Implementation**:
  + **OrientationManager** is responsible for managing the campus map, allowing users to choose two buildings and find the shortest path between them.
  + **Graph** (an ADT) represents the campus map, storing buildings and their distances from one another.
  + **Buildings** are modeled as strings in the **Graph**.
* **ADTs Involved**:
  + **Graph**: This ADT models the campus as a graph where each building is a vertex and the distance between buildings is represented as edges in an adjacency matrix.
  + **Node**: Used within the **Graph** and **MyLinkedList** ADT to represent elements in a list or graph structure. In **Graph**, **Node** objects are part of the graph's connectivity.

**Feature Flow**

1. **Setup Campus Map**:
   * **OrientationManager** creates a **Graph** with six buildings.
   * It uses **Graph**'s addBuilding() and connectBuildings() methods to set up the building names and their connections.
2. **Menu for User**:
   * **OrientationManager** prompts the user to select two buildings (start and end) from a list of campus buildings.
   * The user is then presented with the option to find the shortest path between the two selected buildings.
3. **Shortest Path Calculation**:
   * **OrientationManager** calls the findShortestPath() method of **Graph** to compute the shortest path between the selected buildings using **Dijkstra's Algorithm**.
4. **Display Path**:
   * The shortest path is printed, showing the sequence of buildings from the start to the end, along with the total distance.

**ADT Tracing for OrientationManager**

Let's trace the ADTs involved, their operations, and the flow of data:

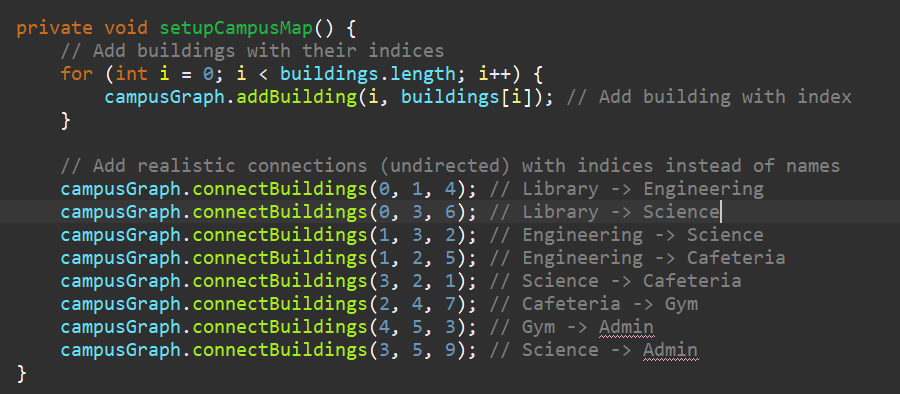
**1. OrientationManager (Feature Implementation)**

* **Attributes**:
  + Graph campusGraph: This ADT models the campus map with buildings as vertices and distances as edges.
  + Scanner scanner: Used to capture user input for building selection.
* **Operations**:
  + orientationMenu(): Displays the menu, prompts the user to select start and end buildings, and calculates the shortest path.

A screen shot of a computer program

AI-generated content may be incorrect.

* + setupCampusMap(): Initializes the **Graph** with buildings and their connections (edges).



* + getChoice(String prompt): Helper method to ensure valid input from the user.

A screen shot of a computer program

AI-generated content may be incorrect.

**2. Graph (Campus Map Model)**

* **Attributes**:
  + String[] buildings: Array holding the names of the buildings (vertices).
  + int[][] adjacencyMatrix: Matrix representing the distances (edges) between buildings.
  + int numVertices: Number of buildings (vertices).
* **Operations**:
  + addBuilding(int index, String buildingName): Adds a building to the graph at a specified index.

A screen shot of a computer code

AI-generated content may be incorrect.

* + connectBuildings(int from, int to, int distance): Connects two buildings with a bidirectional edge, represented by their distance.

A computer screen shot of a program

AI-generated content may be incorrect.

* + findShortestPath(int start, int end): Computes the shortest path between two buildings using **Dijkstra’s Algorithm**.

A screen shot of a computer code

AI-generated content may be incorrect.

* + minDistance(int[] distances, boolean[] visited): Helper method to find the building with the minimum distance that hasn’t been visited.

A computer screen shot of a program code

AI-generated content may be incorrect.

* + printPath(int[] previous, int start, int end): Helper method to print the shortest path between two buildings.

A screen shot of a computer program

AI-generated content may be incorrect.

* + printGraph(): Prints the adjacency matrix representing the campus graph.

A computer screen shot of a code

AI-generated content may be incorrect.

**3. Node (Underlying structure for Graph and MyLinkedList)**

* **Attributes**:
  + T data: Holds the data (building name or other relevant information).
  + Node<T> next: Points to the next node in the list or graph.
* **Operations**:
  + Used for linked list and graph traversal. It’s a fundamental building block for the **Graph** and **MyLinkedList** ADTs.

**Feature 3 Analysis**

* **Graph Representation**: The **Graph** ADT models the campus buildings and their connections. The adjacency matrix makes it easy to represent distances between buildings, allowing quick lookup during the shortest path calculation.
* **Dijkstra’s Algorithm**: The **findShortestPath()** method in the **Graph** class uses **Dijkstra's Algorithm** to compute the shortest path between two buildings, ensuring efficiency and correctness.
* **User Input Validation**: Input is thoroughly validated. If the user selects an invalid building or provides non-integer input, the system will re-prompt the user, ensuring smooth interaction.
* **Campus Navigation**: This feature meets the user's need for guidance by calculating the shortest route between buildings, saving time for students navigating the campus.

This feature provides a solution for navigating the campus by finding the shortest path between buildings. The **Graph** ADT is a perfect fit for representing campus buildings and their interconnections. **Dijkstra’s Algorithm** efficiently calculates the shortest path, while **OrientationManager** handles user interaction and input validation.

**Feature 4: Undo/Redo**

**Big Picture: Relation Between ADTs, Feature, and Models**

**Feature 4** provides a functionality that allows users to undo and redo their actions in the system, such as booking a room. This helps maintain flexibility, allowing users to correct mistakes or reapply changes.

* **Feature Implementation**:
  + **UndoRedoManager** is responsible for managing the undo and redo stacks. It allows users to "undo" (reverse an action) or "redo" (reapply a previously undone action) their actions.
  + **MyStack** is used to store the actions in the form of **BookingRequest** objects. It provides the underlying stack mechanism to manage undo and redo operations.
  + **BookingRequest** is the model class representing the room booking request, which is tracked and managed throughout the undo and redo processes.
* **ADTs Involved**:
  + **MyStack**: A stack-based ADT that stores **BookingRequest** objects in a Last-In-First-Out (LIFO) order. It’s used to store and manage actions in the undo and redo operations.
  + **MyLinkedList**: Used internally by **MyStack** to store the elements (i.e., **BookingRequest** objects).

**Feature Flow**

1. **Adding an Action**:
   * When a booking request is made, the **RoomBookingManager** adds the **BookingRequest** object to the undo stack using the **addAction()** method in **UndoRedoManager**.
   * The redo stack is cleared whenever a new action is added, ensuring that previously redone actions are invalidated when a new action occurs.
2. **Undo Operation**:
   * The user can invoke the **undo()** method in **UndoRedoManager**, which pops the most recent action from the undo stack, pushes it to the redo stack, and returns the undone action.
   * If there are no actions to undo (empty stack), the system informs the user.
3. **Redo Operation**:
   * The user can invoke the **redo()** method in **UndoRedoManager**, which pops the most recent action from the redo stack, pushes it back to the undo stack, and reapplies the action.
   * If there are no actions to redo (empty stack), the system informs the user.
4. **Displaying Undo and Redo Stacks**:
   * The user can view the actions stored in the undo and redo stacks via the **showUndoStack()** and **showRedoStack()** methods, respectively.

**ADT Tracing for Undo/Redo Feature**

Let’s trace the ADTs involved, their operations, and the flow of data:

**1. UndoRedoManager (Feature Implementation)**

* **Attributes**:
  + MyStack<BookingRequest> undoStack: Stack to store actions that can be undone.
  + MyStack<BookingRequest> redoStack: Stack to store actions that have been undone and can be redone.
* **Operations**:
  + addAction(BookingRequest request): Adds a **BookingRequest** to the undo stack, clearing the redo stack.

A screen shot of a computer

AI-generated content may be incorrect.

* + undo(): Pops the most recent action from the undo stack and pushes it to the redo stack.
  + A screen shot of a computer code

    AI-generated content may be incorrect.
  + redo(): Pops the most recent action from the redo stack and pushes it to the undo stack.

A screen shot of a computer code

AI-generated content may be incorrect.

* + showUndoStack(): Displays the actions in the undo stack.

A screen shot of a computer

AI-generated content may be incorrect.

* + showRedoStack(): Displays the actions in the redo stack.

A screen shot of a computer code

AI-generated content may be incorrect.

**2. MyStack (Used for Undo/Redo Stacks)**

* **Attributes**:
  + MyLinkedList<T> list: A linked list used to implement the stack. It stores the elements (in this case, **BookingRequest** objects).
* **Operations**:
  + push(T data): Adds an element to the top of the stack.
  + pop(): Removes and returns the top element from the stack.
  + peek(): Returns the top element without removing it.
  + isEmpty(): Checks if the stack is empty.
  + display(): Displays all elements in the stack.

A screenshot of a computer program

AI-generated content may be incorrect.

**3. MyLinkedList (Underlying Linked List for Stack)**

* **Attributes**:
  + Node<T> head: Points to the first node in the linked list.
* **Operations**:
  + add(T data): Adds a node with the given data at the end of the list.
  + removeExact(T data): Removes a node that matches the given data.
  + remove(): Removes and returns the first node.
  + peek(): Returns the data of the first node without removing it.
  + display(): Displays all the nodes in the list.
  + isEmpty(): Checks if the list is empty.

**4. BookingRequest (Action Model)**

* **Attributes**:
  + String studentName: Name of the student requesting the room.
  + String roomNumber: Room number being requested.
  + int priority: Priority of the room booking.
* **Operations**:
  + getPriority(): Returns the priority of the booking request.
  + toString(): Returns a string representation of the booking request.

A screen shot of a computer program

AI-generated content may be incorrect.

**Feature 4 Analysis**

* **Undo/Redo Logic**: The **UndoRedoManager** leverages the **MyStack** ADT to manage the undo and redo operations. **MyStack** uses a linked list to store the actions (e.g., **BookingRequest** objects), ensuring that operations can be performed efficiently using LIFO (Last In, First Out) order.
* **Action Tracking**: Each booking request is tracked and stored in the **undoStack** and **redoStack**, allowing users to reverse or reapply actions as needed. This feature enhances flexibility and provides error correction.
* **User Input Validation**: Input is validated, ensuring that only valid booking requests are added, with error messages displayed for invalid inputs.
* **Meeting User Needs**: This feature provides users with a safety net, allowing them to undo or redo actions, which is particularly useful when managing critical tasks like room bookings. This is ideal for systems requiring flexibility, such as campus management systems.

This feature enhances the user experience by providing an undo/redo functionality for actions like room bookings. By using **MyStack** and **MyLinkedList**, it efficiently manages the state of user actions and enables smooth rollback and reapplication. **UndoRedoManager** orchestrates the entire process, ensuring that the user's actions can be reversed or reapplied seamlessly.

**Feature 5: Event Participation Registration**

**Big Picture: Relation Between ADTs, Feature, and Models**

**Feature 5** allows students to register for events and tracks their registrations in the order they were received. The key components involved are:

* **EventRegistrationManager**: This feature class manages the event registration process. It interacts with the **MyQueue** ADT to handle the queue of student participants.
* **MyQueue**: This queue-based ADT holds the list of **EventParticipant** objects and handles operations such as adding participants and removing them in the correct order.
* **MyLinkedList**: The **MyQueue** ADT uses **MyLinkedList** as the underlying data structure for storing participants. The linked list ensures that the elements are stored and accessed in a sequential manner.
* **EventParticipant**: This model represents each student who registers for an event. It holds attributes like the student's name and ID, and provides a toString method for displaying the participant details.

**Feature Flow**

1. **Registering a Student**:
   * When a student registers, their **EventParticipant** object is created with the provided name and student ID.
   * The **EventParticipant** object is then enqueued into the **MyQueue** using the enqueue() method.
2. **Admitting the Next Student**:
   * The **admitNext()** method is used to remove (dequeue) the first registered student from the queue and admit them to the event.
   * If the queue is empty, the system informs the user that no students are waiting.
3. **Viewing the Registration Queue**:
   * The user can view all the students waiting to be admitted by using the display() method of the **MyQueue** ADT. This displays all the students in the queue.
4. **Error Handling**:
   * The system ensures that only valid student names and IDs are entered during registration.
   * The user is prompted if invalid choices are made when interacting with the menu options (e.g., entering a non-integer choice).
   * If a student attempts to register after the queue is empty, the system handles this by displaying an appropriate message.

**ADT Tracing for Event Participation Registration**

Let’s trace the ADTs involved, their operations, and the flow of data:

**1. EventRegistrationManager (Feature Implementation)**

* **Attributes**:
  + MyQueue<EventParticipant> queue: A queue that holds the event participants in the order they are registered.
  + Scanner scanner: Used for taking user input.
* **Operations**:
  + eventMenu(): The main method that displays the event registration menu and handles user choices.

A screen shot of a computer screen

AI-generated content may be incorrect.

* + registerStudent(): Registers a student by creating an **EventParticipant** object and adding it to the queue.

A screen shot of a computer code

AI-generated content may be incorrect.

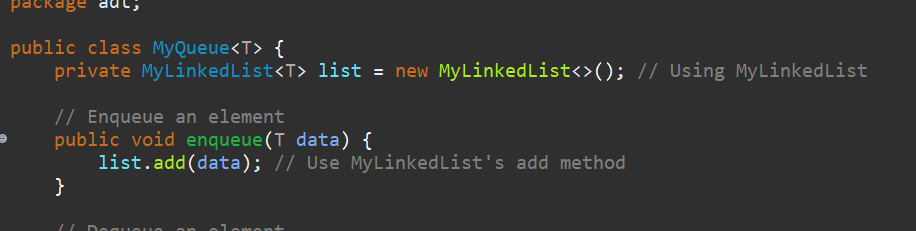
* + admitNext(): Removes the next student from the queue and admits them.

A screen shot of a computer code

AI-generated content may be incorrect.

**2. MyQueue (Event Registration Queue)**

* **Attributes**:
  + MyLinkedList<T> list: A linked list used to implement the queue. It stores **EventParticipant** objects.
* **Operations**:
  + enqueue(T data): Adds an element to the end of the queue.



* + dequeue(): Removes and returns the front element of the queue.

A computer code with text

AI-generated content may be incorrect.

* + isEmpty(): Checks if the queue is empty.

A screenshot of a computer

AI-generated content may be incorrect.

* + display(): Displays all the elements in the queue.

A screen shot of a computer code

AI-generated content may be incorrect.

**3. MyLinkedList (Underlying Linked List for Queue)**

* **Attributes**:
  + Node<T> head: The first node in the linked list.
* **Operations**:
  + add(T data): Adds a node with the given data at the end of the list.
  + removeExact(T data): Removes a node based on its value.
  + remove(): Removes and returns the first node.
  + peek(): Returns the data of the first node without removing it.
  + display(): Displays all the nodes in the list.
  + isEmpty(): Checks if the list is empty.

**4. EventParticipant (Action Model)**

* **Attributes**:
  + String name: The name of the student.
  + String studentID: The ID of the student.
* **Operations**:
  + toString(): Returns a string representation of the **EventParticipant**, displaying the student's name and ID.

A computer screen shot of code

AI-generated content may be incorrect.

**Feature 5 Analysis**

* **Queue Logic**: The **MyQueue** ADT is used to maintain the order of student registrations. It ensures that the first student to register is the first one to be admitted (FIFO - First In, First Out).
* **Event Management**: **EventRegistrationManager** uses **MyQueue** to track participants, and **EventParticipant** models the student registration details (name, student ID).
* **User Input Validation**: Input validation ensures that the user can only enter valid student names, IDs, and choices. This prevents errors and improves the user experience.
* **Meeting User Needs**: This feature meets the basic needs of event registration by allowing users to register students, admit them in order, and view the current list of registered participants. The queue ensures that students are processed fairly and in the correct order.

This feature provides a robust mechanism for registering and managing event participants. By leveraging **MyQueue** and **MyLinkedList**, the system ensures that students are admitted in the correct order based on their registration time. **EventRegistrationManager** orchestrates the entire registration process, while **EventParticipant** models the data for each participant.

If you’re ready, we can move on to **Feature 6: Student Locker Allocation**!

**Feature 6: Student Locker Allocation**

**Big Picture: Relation Between ADTs, Feature, and Models**

**Feature 6** focuses on the management of student lockers. The feature allows for the registration, lookup, and removal of lockers based on student IDs. The key components involved in this feature are:

* **LockerManager**: This feature class manages all interactions related to locker allocation. It allows users to register a student for a locker, lookup a locker by student ID, remove lockers, and view all locker allocations.
* **MyBST**: A Binary Search Tree (BST) is used to store the lockers. Lockers are inserted and retrieved based on student IDs, which ensures that lockers can be efficiently found and managed. The **MyBST** class provides methods for insertion, searching, and deletion of lockers.
* **Locker**: The **Locker** model represents a student's locker. It contains the student’s ID and the locker number, which are essential for registration and lookup.

**Feature Flow**

1. **Registering a Locker**:
   * The **registerLocker()** method prompts the user to input a student ID and locker number. Both values are validated to ensure they are integers.
   * A new **Locker** object is created and inserted into the **MyBST** using the insert() method, which places the locker into the tree based on the student ID.
2. **Looking Up a Locker**:
   * The **lookUpLocker()** method allows the user to search for a locker by student ID. The system uses the search() method of the **MyBST** to find the locker corresponding to the given student ID.
3. **Removing a Locker**:
   * The **removeLocker()** method allows the user to delete a locker by student ID. The **MyBST**'s delete() method is used to remove the locker from the tree.
4. **Viewing All Locker Allocations**:
   * The **lockerMenu()** method has an option to view all the lockers allocated. This is done by calling the inorder() method of the **MyBST** class, which performs an in-order traversal of the tree and prints all the lockers.
5. **Error Handling**:
   * The system handles errors in two key areas: when invalid data (non-integer student ID or locker number) is entered, and when trying to lookup or remove a locker that doesn't exist.
   * Invalid inputs are caught using exception handling, and appropriate messages are displayed to the user.

**ADT Tracing for Student Locker Allocation**

Let’s trace the ADTs involved, their operations, and the flow of data:

**1. LockerManager (Feature Implementation)**

* **Attributes**:
  + MyBST lockerBST: A Binary Search Tree (BST) that stores **Locker** objects, ensuring efficient lookups, insertions, and deletions.
  + Scanner scanner: Used for taking user input.
* **Operations**:
  + lockerMenu(): Displays the locker allocation menu and handles user choices.

A screen shot of a computer program

AI-generated content may be incorrect.

* + registerLocker(): Registers a new locker by taking student ID and locker number as input and inserting it into the **MyBST**.

A computer screen shot of code

AI-generated content may be incorrect.

* + lookUpLocker(): Searches for a locker based on the student ID.

A screen shot of a computer code

AI-generated content may be incorrect.

* + removeLocker(): Deletes a locker based on the student ID.

A screen shot of a computer program

AI-generated content may be incorrect.

**2. MyBST (Binary Search Tree for Locker Management)**

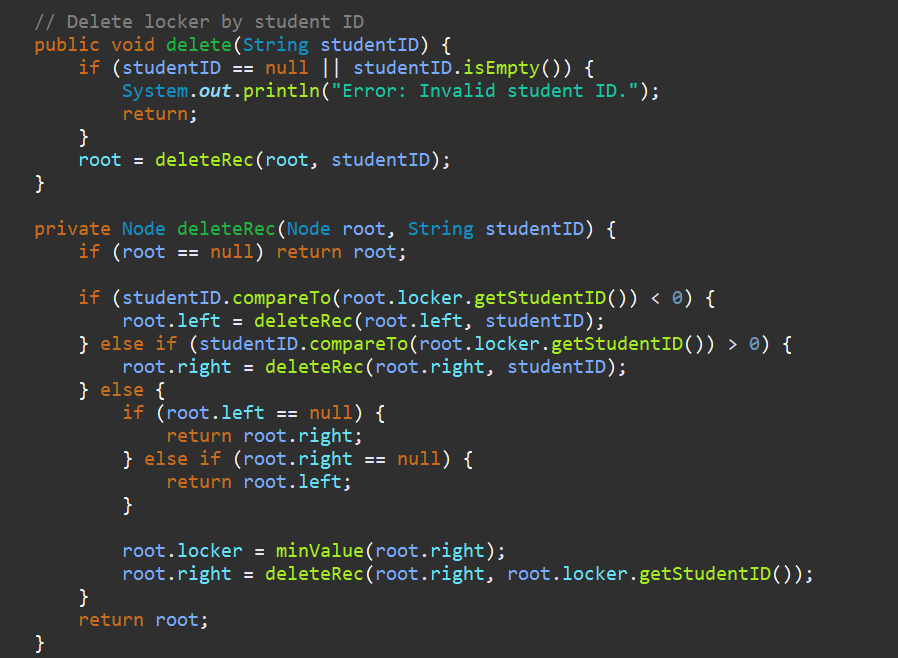
* **Attributes**:
  + Node root: The root node of the BST, which starts the tree structure.
* **Operations**:
  + insert(Locker locker): Inserts a **Locker** object into the tree based on the student ID.

A screen shot of a computer program

AI-generated content may be incorrect.

* + search(String studentID): Searches for a **Locker** object based on the student ID.
  + A screen shot of a computer code

    AI-generated content may be incorrect.
  + delete(String studentID): Deletes a **Locker** object based on the student ID.



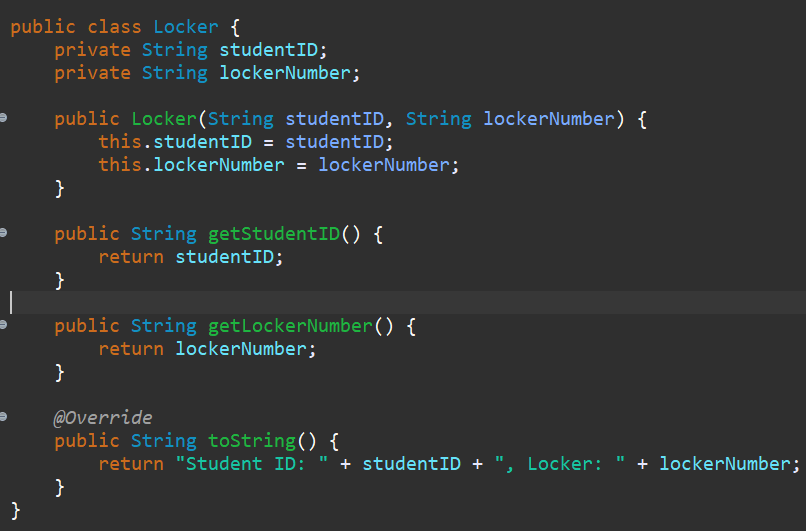
* + inorder(): Traverses the tree in in-order and prints the locker information.

A screen shot of a computer code

AI-generated content may be incorrect.

**3. Locker (Locker Model)**

* **Attributes**:
  + String studentID: The student’s ID.
  + String lockerNumber: The locker number assigned to the student.
* **Operations**:
  + toString(): Returns a string representation of the locker, showing the student ID and locker number.



**Feature 6 Analysis**

* **BST Logic**: The **MyBST** ADT is used to efficiently store lockers, enabling fast lookups, insertions, and deletions based on student IDs. The use of a Binary Search Tree ensures that these operations are efficient (O(log n) time complexity for insertions, deletions, and lookups).
* **Locker Management**: The **LockerManager** class handles all locker-related operations and acts as the interface for the user. It integrates the **MyBST** to manage the locker data and provides methods for registering, removing, and searching lockers.
* **User Input Validation**: The system ensures that student IDs and locker numbers are valid integers through exception handling, preventing invalid input from corrupting the system.
* **Meeting User Needs**: This feature fulfills the basic needs of locker allocation by providing the ability to assign lockers to students, view current allocations, and remove lockers when necessary. The use of **MyBST** ensures efficient management of lockers, while the user interface allows for simple interactions with the system.

Feature 6 leverages **MyBST** to efficiently handle locker allocation and management, ensuring that lockers are assigned, searched, and removed quickly. The **LockerManager** orchestrates the feature and interacts with the **MyBST** to store and manage **Locker** objects, providing a smooth experience for users.

This concludes **Feature 6: Student Locker Allocation**. If you're ready, we can move on to the next feature or any other aspect of the system!

three benefits of using the selected independent ADTs :

**Improved Efficiency**: ADTs like **MyBST** and **MyPriorityLinkedList** provide faster operations (e.g., search, insertion, and deletion) with time complexities like **O(log n)**, which boosts performance, especially for features like locker allocation and room booking.

**Encapsulation and Abstraction**: ADTs hide the complexity of data handling, allowing you to focus on core functionality. This makes the system easier to maintain, modify, and understand without worrying about the internal data structures.

**Scalability and Flexibility**: Independent ADTs provide flexibility to easily extend or modify the system, enabling the addition of new features without significant code changes. This makes the system more adaptable to future requirements.

# 3. Critical evaluate the complexity of each implemented algorithm and ADT in the proposed solution.

| **ADT/Algorithm** | **Operation** | **Time Complexity** | **Explanation** | **Evaluation** |
| --- | --- | --- | --- | --- |
| **MyBST** | Insert | **O(log n)** (avg), **O(n)** (worst) | Insertion in a balanced tree is log(n), but unbalanced tree gives linear time. | Efficient if balanced; worst case **O(n)** in unbalanced tree. |
|  | Search | **O(log n)** (avg), **O(n)** (worst) | Search is log(n) in balanced tree, linear in unbalanced tree. | Efficient for searching but can degrade with unbalanced trees. |
|  | Delete | **O(log n)** (avg), **O(n)** (worst) | Similar to insert, it’s log(n) in balanced trees, linear in unbalanced. | Same as insert; efficiency relies on balance. |
| **MyPriorityLinkedList** | Insert | **O(n)** | Insert involves finding correct position, which takes linear time. | Insertion can be slow with large datasets, but removal is fast (O(1)). |
|  | Remove Highest Priority | **O(1)** | The highest priority element is always at the front of the list. | Very efficient for removing highest priority elements. |
|  | Remove Booking Request | **O(n)** | Requires linear search to find and remove the request. | Efficient for the highest priority, but removal by request takes time. |
| **MyQueue** | Enqueue | **O(1)** | Adding to the end of the queue is constant time. | Highly efficient for adding elements. |
|  | Dequeue | **O(1)** | Removing from the front of the queue is constant time. | Efficient for removing elements. |
|  | Display | **O(n)** | Displays all elements in the queue. | **O(n)** for displaying, as all elements need to be shown. |
| **MyStack** | Push | **O(1)** | Adding an element to the top is constant time. | Efficient for pushing elements. |
|  | Pop | **O(1)** | Removing the top element is constant time. | Efficient for popping elements. |
|  | Peek | **O(1)** | Viewing the top element is constant time. | Efficient for peeking at the top element. |
|  | Display | **O(n)** | Displays all elements in the stack. | **O(n)** for displaying, as all elements need to be shown. |
| **Graph (Dijkstra’s Algorithm)** | Add Building | **O(1)** | Adding a building is a simple array assignment. | Fast operation, constant time. |
|  | Connect Buildings | **O(1)** | Modifying the adjacency matrix takes constant time. | Fast for connecting buildings. |
|  | Find Shortest Path | **O(V^2)** | Dijkstra’s algorithm with adjacency matrix is quadratic in the number of vertices. | Inefficient for large graphs due to **O(V^2)** complexity. Consider alternative graph representations. |
| **MyLinkedList** | Add | **O(n)** | Adding an element involves traversing the list to the end. | Slow for large lists. Better for quick lookups and removals at the head. |
|  | Remove | **O(n)** | Removal requires linear search for specific element. | Slow for large lists due to the linear search. |
|  | Display | **O(n)** | Every element in the list must be visited. | **O(n)** for displaying all elements. |
| **MyLinkedListWithId** | Remove by ID | **O(n)** | Requires a linear search to find and remove by ID. | Slow for large lists, linear search for matching ID. |
| **UndoRedoManager** | Add Action | **O(1)** | Pushing an action to the stack is constant time. | Highly efficient for adding actions to undo stack. |
|  | Undo | **O(1)** | Popping from the undo stack is constant time. | Efficient for undoing actions. |
|  | Redo | **O(1)** | Popping from the redo stack is constant time. | Efficient for redoing actions. |
|  | Show Undo/Redo Stack | **O(n)** | Displaying the stacks takes linear time as all elements need to be shown. | **O(n)** for displaying, as all actions need to be displayed. |

# 4. For feature #3, illustarte step by step in details the implemented algorithms.

**Step-by-Step Breakdown of Dijkstra's Algorithm in the Graph Class**

**1. Graph Class Initialization**

* **Purpose**: The Graph class is used to represent a network of buildings on the campus with their distances (edges).
* **Components**:

buildings[]: An array holding names of buildings.

adjacencyMatrix[][]: A 2D matrix storing the distances between buildings (edges of the graph).

numVertices: Number of buildings (or vertices) in the graph.

* **Algorithm Setup**:

The Graph(int numVertices) constructor initializes the buildings[] and adjacencyMatrix[][] based on the number of buildings.

**2. addBuilding()**

* **Purpose**: Adds a building to the graph.
* **Operation**:

It checks whether the building index is valid and then stores the building name at the given index.

* **Example**: Adds "Library", "Cafeteria", "Gym", etc.

**3. connectBuildings()**

* **Purpose**: Connects two buildings with a specified distance (bi-directional edge).
* **Operation**:

The method checks if the building indices are valid.

It then updates the adjacency matrix to represent the distance between the two buildings.

The adjacency matrix is symmetric (since the connection is bi-directional).

* **Example**: Connects the "Library" to "Engineering" with a distance of 4.

**4. findShortestPath()**

* **Purpose**: Finds the shortest path from a start building to an end building using **Dijkstra’s algorithm**.
* **Steps**:

**Step 1: Initialization**

The distances[] array is initialized with **infinity (Integer.MAX\_VALUE)** for all buildings, except for the starting building, which is set to 0 (distance to itself).

previous[] is initialized with **-1** for all buildings (used to track the path).

visited[] is initialized as false for all buildings (used to check if a building has been processed).

**Step 2: Main Dijkstra Loop**

The algorithm runs for **numVertices - 1 iterations** (since there are numVertices buildings, but we only need numVertices - 1 edges for a connected graph).

In each iteration, we select the building with the smallest tentative distance that hasn’t been visited yet. This is done by calling the helper method minDistance(), which scans the distances[] array and returns the index of the building with the smallest distance that hasn't been visited.

**Step 3: Update Distances**

For each unvisited building, the algorithm checks if the current building can offer a shorter path to any other building. If so, it updates the distances[] and sets the previous[] to the current building. This step is the core of the algorithm, as it ensures that the shortest path is calculated progressively for all buildings.

**Step 4: Path Reconstruction**

After all buildings have been visited and distances have been updated, we can trace the shortest path from the start to the end building using the previous[] array.

The printPath() method recursively traces from the end building back to the start, printing the shortest path.

* **Example**: If the start building is "Library" and the end building is "Admin", the algorithm computes the shortest path between them by checking the adjacency matrix and iterating through the buildings.

**5. minDistance()**

* **Purpose**: A helper function to find the building with the smallest tentative distance that has not been visited yet.
* **Operation**:

It loops through all the buildings and returns the index of the building with the minimum distance that has not yet been visited.

**6. printPath()**

* **Purpose**: Traces the shortest path from the start building to the end building using the previous[] array.
* **Operation**:

It starts from the end building and traces backward to the start, constructing the path.

The path is then printed as a sequence of building names.

**7. Complexity of Dijkstra’s Algorithm**

* **Time Complexity**:

**O(V^2)** (where V is the number of vertices/buildings) because the algorithm uses a simple adjacency matrix (a 2D array) to represent the graph, and finding the minimum distance in each iteration requires scanning all vertices.

If an adjacency list representation was used with a priority queue, the time complexity could be improved to **O(E log V)**.

* **Space Complexity**:

**O(V + E)**, where **V** is the number of vertices and **E** is the number of edges. The adjacency matrix consumes **O(V^2)** space, but other arrays (distances, previous, visited) consume **O(V)** space.

**Final Step: Output**

* **Output**: After running the algorithm, it prints the shortest path (as a sequence of building names) and the total distance.

**Example Output**:

* Shortest path: Library -> Engineering -> Cafeteria -> Gym -> Admin
* Total distance: 18

**nformation Hiding and Encapsulation in the Implemented ADTs**

**What is Information Hiding and Encapsulation?**

* **Encapsulation** is a fundamental concept in Object-Oriented Programming (OOP) where the internal state of an object is protected from outside interference and misuse. It involves bundling the data (attributes) and methods (operations) that manipulate the data into a single unit called a class. This ensures that an object's internal data is accessed and modified only through defined methods, allowing control over how the data is accessed and changed.
* **Information Hiding** refers to the practice of restricting access to certain details of an object’s implementation. By making some attributes and methods private or protected, an object’s internal workings are hidden from the outside world. This helps reduce complexity, protect data integrity, and provide a clear interface for interacting with the object.

**How Information Hiding and Encapsulation Helped in the Implemented ADTs**

In our implemented ADTs (such as MyLinkedList, MyBST, MyQueue, and others), **information hiding and encapsulation** have been used extensively, and here's how they helped in managing the solution:

1. **Data Integrity and Control**:
   * **Encapsulation** ensures that the data within the ADTs (e.g., Node, Locker, BookingRequest) is protected. For example, in MyBST, the Node class encapsulates the Locker object, ensuring that the Locker's internal state is not directly modified by other parts of the program. Instead, the ADT exposes methods like insert, search, and delete, which allow controlled access to the Locker data.
   * **Information hiding** protects sensitive data. For instance, in MyQueue, the internal list (MyLinkedList) that stores data is hidden from direct access by making it private. External entities cannot modify the list directly but must go through the public methods enqueue and dequeue to interact with the data, ensuring that the data is modified only in the intended way.
2. **Simplification and Abstraction**:
   * By hiding the internal details of the data structures (like the MyStack or MyQueue), encapsulation allows the user to interact with these structures in a simplified manner. The user doesn’t need to know how the data is stored (e.g., whether a MyStack uses a linked list or an array internally); they only need to understand the public methods such as push, pop, and peek.
   * For example, when interacting with the MyBST class, the user only needs to call methods like insert, search, and delete without being concerned about the tree's structure (such as how the nodes are arranged or how the insertions are handled).
3. **Error Handling**:
   * **Encapsulation** also plays a role in error handling by centralizing the logic within the ADTs. For instance, in the MyBST class, operations like insert and search validate the input data (e.g., ensuring the student ID isn’t null) and ensure the data is correctly processed. The internal logic for handling errors (e.g., printing an error message or returning a null value) is hidden from the user and encapsulated in the ADT, providing a cleaner, more reliable interface.
   * **Information hiding** reduces the complexity of error handling. External parts of the code do not need to worry about how the data is stored or how exceptions are handled. They can focus on interacting with the ADTs through well-defined interfaces.
4. **Flexibility and Maintainability**:
   * **Encapsulation** allows you to modify the internal implementation of an ADT without affecting the rest of the program. For instance, if you decide to change the internal implementation of MyLinkedList from a singly linked list to a doubly linked list, the external code interacting with the MyQueue or MyStack does not need to change. The interface remains the same, but the implementation can evolve.
   * **Information hiding** enables flexibility in modifying the internals of a class without exposing these changes to the outside world. This makes the system easier to maintain and scale over time.

# 5. In reference to your solution, explain how information hiding and encapuslation helped you when

# using the implemented ADTs. Discuss with justification the view that ADTs are a basis for OOP.

**ADTs as a Basis for Object-Oriented Programming**

**Abstract Data Types (ADTs)** provide a foundation for **Object-Oriented Programming (OOP)** because they encapsulate the key principles of OOP:

1. **Encapsulation**:

ADTs inherently support encapsulation by grouping data and the operations that manipulate the data into a single unit (the class). In OOP, classes are the primary means of achieving encapsulation. ADTs are implemented as classes that provide specific methods to interact with the data, and their internal details are hidden from the user.

For example, in the MyBST class, the insert, search, and delete methods operate on the internal Node and Locker objects. The internal structure of the tree is hidden, and external entities only interact with the tree through these methods.

1. **Abstraction**:

ADTs abstract away the complexity of how data is stored and manipulated. In OOP, abstraction allows users to interact with objects through a simplified interface, without needing to know the underlying details. For example, users of the Graph class do not need to understand how the adjacency matrix is implemented or how the shortest path is computed — they only need to use the findShortestPath method.

In OOP, ADTs provide a means of modeling real-world entities with high-level, abstract concepts. For example, Locker in the MyBST ADT represents a physical locker, and the operations on it are abstracted into methods like insert and search.

1. **Reusability and Modularity**:

ADTs promote **reusability** because once defined, they can be used in multiple parts of the program or even across different projects. For example, the MyLinkedList class can be reused in any situation where a linked list is needed, without needing to reimplement the logic for adding, removing, or searching for elements.

In OOP, ADTs provide modular components that can be assembled into larger systems. The RoomBookingManager uses MyPriorityLinkedList, and LockerManager uses MyBST, demonstrating modularity, where different parts of the system can work together by leveraging ADTs.

1. **Inheritance**:

OOP supports inheritance, which allows new classes to inherit properties and methods from existing classes. While the basic implementation of ADTs in our system does not explicitly use inheritance, many ADTs could be extended or adapted to support new functionality. For example, we could create a specialized version of MyLinkedList with additional features like reverse traversal or thread-safe operations.

1. **Polymorphism**:

ADTs enable polymorphism by allowing different implementations of a data structure to be interchangeable through a common interface. For example, the MyQueue class uses a MyLinkedList internally, but you could easily swap it for another list-based class, such as MyArrayList, without changing the interface used to interact with the queue.

# 6. For each implemented feature using a selected ADT, interpret the trade-off with it's alternatives in

# terms of time and space complexities for all valid operations, support your answer with examples. For

# example, if LinkedList ADT were chosen to implement feature #1, compare it with other possible

# alternative ADTs to impleement same feature such as ArrayList in terms of insert, remove, ..etc.

**Feature 1: Lost & Found (MyLinkedListWithId ADT)**

**Selected ADT: MyLinkedListWithId**

* **Operations**:

**Add**: Add a new item to the list.

**Remove by ID**: Remove an item using its unique ID.

**Display**: Display all items in the list.

**Trade-offs and Comparison with Alternatives:**

1. **LinkedList (MyLinkedList) vs ArrayList (Alternative for MyLinkedListWithId)**:
   * **Insert**:
     + **LinkedList (MyLinkedList)**: Insertions at the head or tail of the list (constant time complexity, O(1)).
     + **ArrayList**: Insertions at the end (O(1) average time), but insertions in the middle or at the beginning require shifting elements (O(n) time complexity).

**Conclusion**: If the LostFoundManager needs frequent insertions or deletions at arbitrary positions (such as the beginning or middle), **LinkedList** is better. **ArrayList** performs well only if elements are added at the end of the list.

1. **Remove by ID**:

**LinkedList**: To remove by ID, we need to traverse the entire list, which has a time complexity of **O(n)** where n is the number of elements.

**ArrayList**: Same as **LinkedList**, but if items are removed from the middle, the elements following the removed item need to be shifted, which also results in **O(n)** time complexity.

**Conclusion**: Both **LinkedList** and **ArrayList** have **O(n)** time complexity for removing elements, so the trade-off here is minimal. However, if the operation to remove an element is more frequent, **LinkedList** offers better flexibility in terms of time complexity for other insertions and deletions.

1. **Space Complexity**:

**LinkedList**: Uses **O(n)** space for storing n elements, and each element has an additional pointer to the next node (increased overhead).

**ArrayList**: Also uses **O(n)** space but stores elements in a contiguous memory block. This results in lower overhead compared to LinkedList.

**Conclusion**: **ArrayList** is more space-efficient because it doesn’t require extra memory for storing pointers like in **LinkedList**.

**Feature 2: Room Booking (MyPriorityLinkedList ADT)**

**Selected ADT: MyPriorityLinkedList**

* **Operations**:

**Insert**: Insert a booking request based on priority.

**Remove Highest Priority**: Remove the highest priority booking.

**Display**: Display all the bookings.

**Trade-offs and Comparison with Alternatives:**

1. **LinkedList (MyLinkedList) vs PriorityQueue (Alternative for MyPriorityLinkedList)**:

**Insert**:

* + - **MyPriorityLinkedList**: Insertions are made in sorted order based on priority. The time complexity is **O(n)** since we must traverse the list to find the right spot.
    - **PriorityQueue**: Insertions have **O(log n)** time complexity because a binary heap is used to maintain the heap property.

**Conclusion**: **PriorityQueue** provides faster insertions compared to **MyPriorityLinkedList**, which performs **O(n)** insertions. For large datasets, **PriorityQueue** is more efficient for handling priority-based operations.

1. **Remove Highest Priority**:

**MyPriorityLinkedList**: To remove the highest priority item (head of the list), it takes **O(1)** time complexity since the highest priority item is at the beginning of the list.

**PriorityQueue**: Removing the highest priority item has **O(log n)** time complexity since the heap needs to be restructured after the removal.

**Conclusion**: For removal operations, **MyPriorityLinkedList** is more efficient with **O(1)** removal compared to **PriorityQueue**’s **O(log n)**. However, the **PriorityQueue** will still perform better in the long run for a large number of insertions and deletions due to its **O(log n)** time complexity.

1. **Space Complexity**:

**MyPriorityLinkedList**: Uses **O(n)** space, with each element being stored in a node containing the data and a pointer.

**PriorityQueue**: Uses **O(n)** space, but the internal structure (binary heap) is more compact compared to a linked list.

**Conclusion**: **PriorityQueue** is slightly more space-efficient as it uses a heap-based structure instead of extra pointers like a linked list. However, the space complexity difference is negligible for most practical use cases.

**Feature 3: Student Orientation Help (Graph ADT)**

**Selected ADT: Graph**

* **Operations**:

**Add Building**: Add a building to the campus map.

**Connect Buildings**: Connect two buildings with a distance.

**Find Shortest Path**: Use Dijkstra’s algorithm to find the shortest path between two buildings.

**Display Graph**: Display the campus map and its connections.

**Trade-offs and Comparison with Alternatives:**

1. **Adjacency Matrix (Graph) vs Adjacency List (Alternative for Graph)**:

**Add Building**:

* + - **Adjacency Matrix**: Adds a building in **O(1)** time complexity.
    - **Adjacency List**: Adding a building takes **O(1)** time as well, but extra space may be needed to maintain lists for each building.

**Conclusion**: The operation for adding buildings is fairly similar in both approaches, so the trade-off is minimal for this operation.

1. **Connect Buildings**:

**Adjacency Matrix**: Connect two buildings (i.e., add an edge) in **O(1)** time because it directly accesses the matrix using row and column indices.

**Adjacency List**: Connect buildings in **O(1)** time if the list is implemented as a doubly-linked list (assuming no sorting is needed).

**Conclusion**: Both methods are efficient for connecting buildings, but **Adjacency Matrix** is generally simpler in implementation for dense graphs, while **Adjacency List** is better for sparse graphs.

1. **Find Shortest Path (Dijkstra’s Algorithm)**:

**Adjacency Matrix**: Running Dijkstra’s algorithm on an adjacency matrix has a time complexity of **O(n²)** because every edge must be checked for every vertex.

**Adjacency List**: Dijkstra’s algorithm on an adjacency list has a time complexity of **O((V + E) log V)** where **V** is the number of vertices and **E** is the number of edges. This is more efficient for sparse graphs because fewer edges need to be checked.

**Conclusion**: **Adjacency List** is generally more efficient for sparse graphs, whereas **Adjacency Matrix** performs better for dense graphs but has higher time complexity in pathfinding operations.

1. **Space Complexity**:

**Adjacency Matrix**: Uses **O(n²)** space, where **n** is the number of buildings. It is inefficient for graphs with a large number of vertices and relatively few edges.

**Adjacency List**: Uses **O(V + E)** space, where **V** is the number of vertices and **E** is the number of edges. This is more space-efficient for sparse graphs.

**Conclusion**: **Adjacency List** is more space-efficient for sparse graphs, while **Adjacency Matrix** may be more efficient for dense graphs, but at the cost of high space complexity.

In summary, each ADT has its advantages and trade-offs, and the choice depends on the specific feature and the operations required:

* **For Feature 1 (Lost & Found)**: The **LinkedList** is optimal for frequent insertions and deletions, especially in the middle of the list. The **ArrayList** would perform worse with such operations.
* **For Feature 2 (Room Booking)**: **PriorityQueue** provides better performance for large datasets due to its **O(log n)** insertion and removal, whereas **MyPriorityLinkedList** offers **O(1)** removal but slower insertions.
* **For Feature 3 (Orientation Help)**: The **Adjacency Matrix** is simpler and performs better for dense graphs, but **Adjacency List** is more space and time-efficient for sparse graphs.