Final Project: Task Management System with Advanced Features

Submission Instructions:

1. **Source Code:** Submit all Java files in a zipped folder. Make sure your code follows all good coding practices - provide useful comments, make appropriate packages and classes.
2. **Output Screenshots:** Test all features and edge cases and capture the output. Include images showing the system in action (e.g., tasks being added, dependencies visualized). Label each screenshot appropriately.
3. **Documentation:** Provide a README file summarizing the project and usage instructions.
4. **Presentation:** Record a 5-minute presentation summarizing the design and functionalities of your code and showcasing 1-2 example outputs. You can screen record with your voice. Upload the 5 minute video.
5. (OPTIONAL) **Resume Addition:** *"Developed a Java-based Task Management System incorporating advanced data structures (e.g., graphs, trees, hash maps) to manage complex task operations like prioritization, dependencies, and analytics."*

Submission file should contain:

1. *Code Zip File*
2. *Folder of all labeled screenshots*
3. *README file*
4. *Presentation Video*
5. *OPTIONAL - Resume with added project*

Grading Criteria:

The final project is worth 10% of your total grade. Below is the grading criteria. This project will be scored for 100 marks.

Criteria	Marks
Correct implementation of features	30
Effective implementation of data structures	30
Code quality and documentation	20
Output	10
Presentation and usability	10

Develop an object-oriented Java application that serves as a comprehensive **Task Management System**. This system will allow users to create, manage, prioritize, and analyze tasks using a variety of advanced data structures. The project will not only solidify your understanding of data structures but also provide a **resume-worthy accomplishment** to showcase during job applications.

Key Features

1. **Task Storage and Retrieval**
    - Use an **Array** to store fixed-size information such as predefined task categories.
    - Implement:
        - **Singly Linked List** to store task history.
        - **Doubly Linked List** for bidirectional navigation of tasks within a category.
        - **Circular Linked List** to manage recurring tasks.
2. **Task Operations**
    - Use a **Stack** to enable undo/redo functionality for task edits.
    - Implement a **Queue** or **Deque** to schedule tasks in FIFO/LIFO order.
    - Use a **Priority Queue** to handle task execution based on urgency or deadlines.
3. **Task Analytics**
    - Build a **Binary Search Tree (BST)** to organize tasks by due dates for efficient searching and traversal.
    - Use a **Binary Heap** to dynamically identify the most or least urgent task.
4. **User Management**
    - Use a **HashSet** to store unique user IDs, preventing duplicate registrations.
    - Implement a **HashMap** to map user IDs to their respective task lists for quick access and management.
5. **Task Dependencies**
    - Represent task dependencies using a **Directed Graph**, where nodes are tasks and edges represent dependencies.
    - Detects and handles circular dependencies using graph traversal techniques (DFS/BFS).
6. **Visualization and Reports**
    - Generate task schedules or dependencies as text-based outputs, such as adjacency lists, adjacency matrices, or tree traversal results.
    - Provide functionality to export task reports (e.g., pending tasks, completed tasks) to a text file for user reference.

Custom Data Structure Implementations Required

The following data structures must be implemented manually to deepen your understanding of their mechanics:

- **Array**: Implement dynamic resizing, and indexing.
- **Linked Lists:** Implement singly, doubly, and circular linked lists, focusing on node creation, insertion, deletion, and traversal.
- **Stack and Queue**: Build simple implementations using arrays or linked lists.
- **Binary Search Tree (BST)**: Implement operations for searching, inserting, deleting, and traversing.
- **Graph**: Represent task dependencies using an adjacency list or matrix. Implement DFS and BFS for traversal.

Use Java's Built-In Classes

The following data structures can leverage Java's standard library for real-world application experience:

- **Priority Queue**: Use PriorityQueue to handle priority-based scheduling.
- **HashSet**: Use HashSet to enforce uniqueness of user IDs.
- **HashMap**: Use HashMap to associate user IDs with their respective task lists.
- **Binary Heap**: Optional—PriorityQueue can be used as an underlying structure.

Step-by-Step Instructions to help you get started

Phase 1: Design the System

1. **Understand the Requirements**:
    - Break down each feature and decide which data structure fits best.
    - List down the relationships between tasks, categories, users, and schedules.
2. **Create** the main classes, such as:
    - **Task** (attributes: id, name, dueDate, priority, dependencies)
    - **User** (attributes: id, name, taskList)
    - **TaskManager** (core functionalities)
    - **GraphManager** (dependency management)
3. **Plan Data Structures**:

- Map each feature to its respective data structure (e.g., PriorityQueue for high-priority tasks).

Phase 2: Implement the System

1. **Set Up the Project**:
   - Create a Java project in your IDE with packages for organization (e.g., data_structures, tasks, managers).
2. **Implement the Custom Data Structures**:
   - Build classes for arrays, linked lists, stacks, queues, trees, and graphs.
   - Test each implementation separately before integrating them into the project.
3. **Integrate Built-In Data Structures**:
   - Use PriorityQueue, HashSet, and HashMap where specified.
4. **Build Core Functionalities**:
   - Add, delete, edit, schedule, and prioritize tasks.
   - Enable undo/redo functionality using a custom stack.
   - Create tree-based methods to analyze and sort tasks by due dates.
   - Add graph-based methods to manage and validate task dependencies.

Phase 3: Develop User Interface

1. **Console-Based Interaction**:
   - Create a console based menu system for users to interact with the system.
   - Options include:
     - Add/Delete/View/Edit tasks.
     - Undo/Redo operations.
     - Export reports and visualize dependencies.
2. **Test Interactions**:
   - Ensure every menu option integrates seamlessly with the underlying data structures.

Phase 4: Final Testing and Documentation

1. **Test Features and Edge Cases**:
   - Handle invalid inputs, circular dependencies, and duplicate entries.

2. **Document the Code**:
    - Add comments and a README file explaining the system design and usage.
3. **Prepare Deliverables**:
    - Include the source code, screenshots of the output, and a brief project description for resumes.