**Class Definition: BankersAlgorithm**

**Private Members:**

1. **numProcesses**: Number of processes in the system.
2. **numResources**: Number of resources in the system.
3. **maximum**: 2D vector representing the maximum demand matrix for each process.
4. **allocation**: 2D vector representing the current allocation matrix for each process.
5. **need**: 2D vector representing the need matrix (difference between maximum and allocation) for each process.
6. **available**: 1D vector representing the available resources in the system.
7. **request**: 2D vector representing the request matrix (requests made by each process).
8. **waitForGraph**: 2D vector representing the wait-for graph used for deadlock detection.

**Private Methods:**

1. **displayMatrices**: Displays the current state of available resources, maximum matrix, allocation matrix, need matrix, and request matrix.
2. **isSafe**:
   * **Purpose**: Checks if the system is in a safe state using the Banker's Algorithm.
   * **Logic**:
     + **Work Vector**: Tracks the available resources during the check.
     + **Finish Vector**: Tracks which processes have been finished.
     + **Safety Check**: Iterates through processes to find a safe sequence by checking if each process can be safely executed with the current available resources.
     + If a safe sequence is found, the system is safe; otherwise, it is unsafe.
3. **detectDeadlock**:
   * **Purpose**: Detects cycles in the wait-for graph using DFS to identify deadlocks.
   * **Logic**:
     + **DFS**: Uses depth-first search to detect cycles in the wait-for graph. A cycle indicates a deadlock.
4. **handleDeadlock**:
   * **Purpose**: Handles detected deadlocks by terminating a process and adding a new process.
   * **Logic**:
     + **Terminate Process**: Iterates through processes to find and remove one that causes the deadlock.
     + **Recalculate Matrices**: Updates matrices and the wait-for graph after process removal.
     + **Add New Process**: Adds a new process with random maximum, allocation, and need values.
     + **Recheck Safety**: Checks if the system is safe after adding the new process.
5. **addNewProcess**:
   * **Purpose**: Adds a new process with random values for maximum, allocation, and need matrices.
   * **Logic**:
     + **Update Matrices**: Updates the maximum, allocation, and need matrices for the newly added process.
     + **Update Wait-for Graph**: Updates the wait-for graph to include the new process.

**Constructor: BankersAlgorithm(int p, int r)**

* **Purpose**: Initializes the Banker's Algorithm instance with p processes and r resources.
* **Logic**: Resizes matrices and vectors to accommodate the number of processes and resources.

**Method: generateRandomMatrices**

* **Purpose**: Generates random values for the available resources, maximum matrix, allocation matrix, request matrix, and recalculates the need matrix.
* **Logic**:
  + **Available Resources**: Randomly assigns values to available resources.
  + **Maximum Matrix**: Randomly assigns maximum demand values for each process.
  + **Allocation Matrix**: Randomly assigns allocation values (up to the maximum demand).
  + **Request Matrix**: Randomly assigns request values (0 or 1).
  + **Need Matrix**: Calculated as the difference between the maximum and allocation matrices.
  + **Update Wait-for Graph**: Based on the request matrix and need matrix, updates the wait-for graph.

**Method: runSafetyCheck**

* **Purpose**: Runs the safety check and handles any detected deadlocks.
* **Logic**:
  + **Safety Check**: Uses isSafe to check if the system is in a safe state.
  + **Deadlock Handling**: If unsafe, uses detectDeadlock to check for deadlocks and handleDeadlock to address them.

**Main Function: main()**

* **Purpose**: Entry point of the program, which sets up and runs the Banker's Algorithm simulation.
* **Logic**:
  + **Input**: Prompts the user to enter the number of processes and resources.
  + **Initialize**: Creates an instance of BankersAlgorithm with the given number of processes and resources.
  + **Generate Random Matrices**: Calls generateRandomMatrices to set up the system with random values.
  + **Run Safety Check**: Calls runSafetyCheck to evaluate the system's safety and handle any deadlocks.

**Example Execution Walkthrough**

1. **Input Number of Processes and Resources**:
   * Example: 3 processes and 3 resources.
2. **Generate Random Matrices**:
   * Randomly populate matrices for maximum demand, allocation, request, and available resources.
3. **Run Safety Check**:
   * **Initial Check**: Determine if the system is in a safe state.
   * **If Unsafe**:
     + Detect deadlocks using the wait-for graph.
     + Terminate a process if necessary.
     + Recalculate and add a new process.
     + Check system safety again.

**Key Points to Emphasize in an Interview**

1. **Banker's Algorithm**:
   * Aims to ensure safe resource allocation and avoid deadlocks.
   * Determines if a system is in a safe state by checking if resources can be allocated without leading to an unsafe state.
2. **Deadlock Detection and Handling**:
   * Uses a wait-for graph and DFS to detect cycles indicating deadlocks.
   * Handles deadlocks by terminating processes and adding new ones, recalculating system matrices and safety.
3. **Random Matrix Generation**:
   * Ensures diverse scenarios and tests various system states.
4. **Dynamic System Modification**:
   * Ability to modify the system (terminate and add processes) dynamically to handle detected issues.

By explaining each part in this manner, you'll provide a clear and comprehensive overview of how the code implements the Banker's Algorithm and manages deadlocks.