PSUDOCODE

```
// Define the main class for solving the Eight Queens puzzle
public class EightQueensSolver {

    // Define a static inner class for representing Queen positions
    static class QueenPosition {
        int row;
        int col;

        // Constructor for QueenPosition
        public QueenPosition(int row, int col) {
            this.row = row;
            this.col = col;
        }
    }

    // Define a static inner class for representing linked list nodes
    static class Node {
        QueenPosition queen;
        Node next;

        // Constructor for Node
        public Node(QueenPosition queen) {
            this.queen = queen;
            this.next = null;
        }
    }

    // Define a static inner class for implementing a linked list stack of QueenPositions
    static class QueenStack {
        Node top;

        // Constructor for QueenStack
        public QueenStack() {
            top = null;
        }

        // Push a QueenPosition onto the stack
        public void push(QueenPosition queen) {
            Node newNode = new Node(queen);
            newNode.next = top;
            top = newNode;
        }

        // Pop and return a QueenPosition from the top of the stack
        public QueenPosition pop() {
            if (top == null) {
                return null;
            }
```

```
            QueenPosition queen = top.queen;
            top = top.next;
            return queen;
        }

        // Peek at the QueenPosition on the top of the stack
        public QueenPosition peek() {
            if (top == null) {
                return null;
            }
            return top.queen;
        }

        // Check if the stack is empty
        public boolean isEmpty() {
            return top == null;
        }
    }

    // Function to check if two QueenPositions conflict (same row, column, or diagonal)
    public static boolean isConflict(QueenPosition q1, QueenPosition q2) {
        return q1.row == q2.row || q1.col == q2.col || Math.abs(q1.row - q2.row) ==
Math.abs(q1.col - q2.col);
    }

    // Function to check if a valid solution is found (eight queens are placed)
    public static boolean isSolution(QueenPosition[] queens) {
        return queens.length == 8;
    }

    // Function to display the chessboard with queens placed
    public static void displayBoard(QueenPosition[] queens) {
        char[][] chessboard = new char[8][8];

        // Mark the positions of queens on the chessboard
        for (QueenPosition queen : queens) {
            chessboard[queen.row][queen.col] = 'Q';
        }

        // Display the chessboard
        for (int row = 0; row < 8; row++) {
            for (int col = 0; col < 8; col++) {
                if (chessboard[row][col] == 'Q') {
                    System.out.print("Q ");
                } else {
                    System.out.print("- ");
                }
            }
            System.out.println();
        }
    }

    // Function to solve the puzzle using linked-list stack approach
    public static void solveWithLinkedListStack() {
```

```
// Create a stack to store queen positions
QueenStack queensStack = new QueenStack();
boolean success = false;

// Start with an initial queen position at (0, 0)
queensStack.push(new QueenPosition(0, 0));

// Continue until a solution is found or the stack is empty
while (!success && !queensStack.isEmpty()) {
    // Get the current queen position from the stack
    QueenPosition current = queensStack.peek();

    // If the current queen's column is 8, backtrack
    if (current.col == 8) {
        queensStack.pop();
        if (!queensStack.isEmpty()) {
            QueenPosition prevQueen = queensStack.pop();
            queensStack.push(new QueenPosition(prevQueen.row, prevQueen.col + 1));
        }
    } else {
        boolean conflict = false;
        // Check for conflicts with previously placed queens
        for (Node node = queensStack.top; node != null; node = node.next) {
            if (node.queen != current && isConflict(node.queen, current)) {
                conflict = true;
                break;
            }
        }

        if (conflict) {
            // If there's a conflict, adjust positions or backtrack
            if (current.col == 7) {
                queensStack.pop();
                if (!queensStack.isEmpty()) {
                    QueenPosition prevQueen = queensStack.pop();
                    queensStack.push(new QueenPosition(prevQueen.row, prevQueen.col + 1));
                }
            } else {
                queensStack.pop();
                queensStack.push(new QueenPosition(current.row, current.col + 1));
            }
        } else {
            // If no conflict, proceed to the next row or column
            if (current.row == 7) {
                success = true;
                break;
            }
            queensStack.push(new QueenPosition(current.row + 1, 0));
        }
    }
}

// Display the solution or indicate no solution
if (success) {
```

```
            QueenPosition[] solution = new QueenPosition[8];
            Node node = queensStack.top;
            int index = 7;
            while (node != null) {
                solution[index--] = node.queen;
                node = node.next;
            }
            displayBoard(solution);
        } else {
            System.out.println("No solution found.");
        }
    }

    // Main function
    public static void main(String[] args) {
        solveWithLinkedListStack();
    }
}
```