

Group Project

Group K



June 16, 2025

Binary Search Tree Class

// Array List is A generic type of Data Structure  
// List is Also a dATA Structure But here We Can Store Only 1 type of A Data S......  
import java.util.ArrayList;  
// for the operation on user names  
  
import java.util.List;  
// For Storing the Usernames  
  
  
public class BinarySearchTree {  
  
 private BSTNode root;  
  
 public BinarySearchTree()  
  
 {  
  
 this.root = null;  
 }  
  
 // Inserts a username into the BST  
 public void insert(String username)  
  
 {  
 root = insertRecursive(root, username);  
 }  
// Using the Recursive Function For the Addition of the Names In the BST  
 private BSTNode insertRecursive(BSTNode current, String username) {  
 if (current == null) {  
 return new BSTNode(username);  
 }  
  
  
 // Compare usernames to decide whether to go left or right  
 int cmp = username.compareTo(current.username);  
  
 if (cmp < 0) {  
  
 current.left = insertRecursive(current.left, username);  
 }  
  
 else if (cmp > 0) {  
 current.right = insertRecursive(current.right, username);  
 }  
 // If cmp == 0, username already exists, do nothing (BSTs typically don't store duplicates)  
 return current;  
 }  
  
 // Deletes a username from the BST  
 public void delete(String username) {  
 root = deleteRecursive(root, username);  
 }  
  
 private BSTNode deleteRecursive(BSTNode current, String username) {  
 if (current == null) {  
 return null; // Username not found  
 }  
  
 int cmp = username.compareTo(current.username);  
 // if the User Val is Less then the 0 then it must be On the Left hand Side of the Tree  
 if (cmp < 0) {  
 current.left = deleteRecursive(current.left, username);  
 } else if (cmp > 0) {  
 current.right = deleteRecursive(current.right, username);  
 }  
  
 else {  
 // Node to be deleted found  
 if (current.left == null && current.right == null) {  
 return null; // Case 1: No children  
 } else if (current.left == null) {  
 return current.right; // Case 2: Only right child  
 } else if (current.right == null) {  
 return current.left; // Case 2: Only left child  
 } else {  
 // Case 3: Two children - find the smallest in the right subtree (inorder successor)  
 String smallestValue = findSmallestValue(current.right);  
 current.username = smallestValue; // Replace current node's value with inorder successor  
 current.right = deleteRecursive(current.right, smallestValue); // Delete the inorder successor  
 }  
 }  
 return current;  
 }  
  
 // Helper to find the smallest value in a subtree (used for deletion with two children)  
 private String findSmallestValue(BSTNode root) {  
 return root.left == null ? root.username : findSmallestValue(root.left);  
 }  
  
 // Performs an inorder traversal to get all usernames in alphabetical order  
 public List<String> inorderTraversal() {  
 List<String> sortedUsernames = new ArrayList<>();  
 inorderRecursive(root, sortedUsernames);  
 return sortedUsernames;  
 }  
  
 // Inorder Printing  
 /\*  
 left  
 Node  
 Right  
  
 \*/  
 private void inorderRecursive(BSTNode node, List<String> sortedUsernames) {  
 if (node != null) {  
 inorderRecursive(node.left, sortedUsernames);  
 sortedUsernames.add(node.username);  
 inorderRecursive(node.right, sortedUsernames);  
 }  
 }  
  
 // Clears the BST (resets it to an empty tree)  
 public void clear() {  
 root = null;  
 }  
}

BST Node Class

public class BSTNode {  
  
 String username;  
  
 BSTNode left;  
  
 BSTNode right;  
  
 public BSTNode(String username) {  
  
 this.username = username;  
 this.left = null;  
 this.right = null;  
  
 }  
  
}  
/\*  
(BST) class is used to store, manage, and organize usernames in a sorted and efficient manner.  
 \*/

CHATAPP

import java.util.ArrayList;  
import java.util.List;  
import java.util.Scanner;  
  
// Basically For the Restriction of the gmail id which is the Built-in the Java  
  
import java.util.regex.Pattern;  
  
// The main application class for the chat application.  
// Handles user interaction, menu navigation, and orchestrates calls to UserDatabase and Message classes.  
public class ChatApp {  
  
 // Static instance of Message to perform message-related operations.  
 static Message *messageObj* = new Message(null, null, null);  
 // Static variable to hold the currently logged-in user.  
 static User *loginUser*;  
  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.***in***); // Scanner for user input.  
 UserDatabase userDatabase = new UserDatabase(); // Manages user accounts and DSA features.  
  
 // Main application loop. Continues until user chooses to exit.  
 while (true) {  
 // Display login/signup menu if no user is currently logged in.  
 if (*loginUser* == null) {  
 System.***out***.println("\n--- Welcome to WhatsApp! ---");  
 System.***out***.println("How Can I Assist You Today?");  
 System.***out***.println("1. Sign Up");  
 System.***out***.println("2. Login");  
 System.***out***.println("3. Exit");  
 System.***out***.print("Please choose an option: ");  
  
 int choice = *getValidatedInput*(scanner); // Get validated integer input from user.  
  
 switch (choice) {  
 case 1: // Sign Up  
 System.***out***.print("Enter a new Username: ");  
 String username = scanner.nextLine();  
  
 // Check if username already exists.  
 if (userDatabase.getUserDatabase().containsKey(username)) {  
 System.***out***.println("Username '" + username + "' is already taken. Please choose a different one.");  
 break;  
 }  
  
 // Validate password strength.  
 String password;  
 while (true) {  
 System.***out***.print("Enter password (at least 4 characters): ");  
 password = scanner.nextLine();  
 if (password.length() < 4) {  
 System.***out***.println("Your password is too weak. Please enter a password with at least 4 characters.");  
 } else {  
 break;  
 }  
 }  
  
 // Validate email format.  
 String email;  
 while (true) {  
 System.***out***.print("Enter Email address: ");  
 email = scanner.nextLine();  
 if (!*isValidEmail*(email)) {  
 System.***out***.println("Invalid Email Address. Please enter a valid email (e.g., user@example.com).");  
 } else {  
 break;  
 }  
 }  
  
 userDatabase.signup(username, password, email); // Attempt to register the new user.  
 break;  
  
 case 2: // Login  
 System.***out***.print("Enter username: ");  
 String loginUsername = scanner.nextLine();  
 System.***out***.print("Enter password: ");  
 String loginPassword = scanner.nextLine();  
 *loginUser* = userDatabase.login(loginUsername, loginPassword); // Attempt to log in the user.  
 break;  
  
 case 3: // Exit  
 System.***out***.println("Exiting the application. Goodbye!");  
 scanner.close(); // Close the scanner before exiting.  
 return; // Terminate the program.  
  
 default:  
 System.***out***.println("Invalid option. Please try again.");  
 }  
 } else {  
 // Display logged-in user menu.  
 System.***out***.println("\n--- Logged-In Menu (" + *loginUser*.getUsername() + ") ---");  
 System.***out***.println("1. Send Message");  
 System.***out***.println("2. View Chats");  
 System.***out***.println("3. View Messages with a User (Means Receivers Name");  
 System.***out***.println("4. View Online Users");  
 System.***out***.println("5. Search Messages (by keyword)");  
 System.***out***.println("6. Delete Account");  
 System.***out***.println("7. View Recent Logins"); // (LinkedList) that Will Traverse Over the Recent Users  
 System.***out***.println("8. View Last N Messages (Reverse Order)"); // That Will Use Stack here First in First Out  
 System.***out***.println("9. View All Users (Alphabetical)"); // (Binary Search Tree)  
 System.***out***.println("10. Logout");  
 System.***out***.println("11. Exit");  
 System.***out***.print("Choose an option: ");  
  
 int choice = *getValidatedInput*(scanner); // Get validated integer input.  
  
 switch (choice) {  
 case 1: // Send Message  
 System.***out***.print("Enter receiver's username: ");  
 String receiverName = scanner.nextLine();  
  
 User receiverUser = userDatabase.getUser(receiverName); // Get recipient's User object.  
  
 if (receiverUser == null) {  
 System.***out***.println("User with username '" + receiverName + "' not found.");  
 break;  
 }  
  
 if (receiverUser.equals(*loginUser*)) {  
 System.***out***.println("You cannot send messages to yourself.");  
 break;  
 }  
  
 System.***out***.print("Enter your message: ");  
 String messageContent = scanner.nextLine();  
 *messageObj*.sendMessage(*loginUser*, receiverUser, messageContent); // Send the message.  
 break;  
  
 case 2: // View Chats  
 System.***out***.println("\n--- Your Chats ---");  
 ArrayList<String> chats = *messageObj*.getChats(*loginUser*); // Get list of chat partners.  
 if (chats.isEmpty()) {  
 System.***out***.println("No chats available. Send a message to start one!");  
 } else {  
 System.***out***.println("You have chats with:");  
 for (String chatUser : chats) {  
 // Display online status of chat partners.  
 System.***out***.println("- " + chatUser + " (" + (userDatabase.getUser(chatUser).isOnline() ? "Online" : "Offline") + ")");  
 }  
 }  
 break;  
  
 case 3: // View Messages with a User  
 System.***out***.print("Enter username to view messages: ");  
 String chatUsername = scanner.nextLine();  
 User chatUser = userDatabase.getUser(chatUsername); // Get the chat partner's User object.  
 if (chatUser == null) {  
 System.***out***.println("User with username '" + chatUsername + "' not found.");  
 break;  
 }  
 System.***out***.println("\n--- Chat with " + chatUsername + " ---");  
 ArrayList<Message> messages = *messageObj*.getMessages(*loginUser*, chatUser); // Get all messages, chronologically sorted.  
 if (messages.isEmpty()) {  
 System.***out***.println("No messages with " + chatUsername + ".");  
 } else {  
 for (Message message : messages) {  
 message.displayMessage(); // Display each message.  
 }  
 }  
 break;  
  
 case 4: // View Online Users  
 System.***out***.println("\n--- Online Users ---");  
 boolean foundOnlineUser = false;  
 // Iterate through all users to find who is online (excluding the current user).  
 for (User user : userDatabase.getUserDatabase().values()) {  
 if (user.isOnline() && !user.equals(*loginUser*)) {  
 System.***out***.println("- " + user.getUsername());  
 foundOnlineUser = true;  
 }  
 }  
 if (!foundOnlineUser) {  
 System.***out***.println("No other users are currently online.");  
 }  
 break;  
  
 case 5: // Search Messages (by keyword)  
 System.***out***.print("Enter keyword to search for: ");  
 String keyword = scanner.nextLine().toLowerCase(); // Convert keyword to lowercase for case-insensitive search.  
 System.***out***.println("\n--- Search Results for '" + keyword + "' ---");  
 boolean foundMessages = false;  
 // Iterate through all users and their messages to find matching content.  
 for (User user : userDatabase.getUserDatabase().values()) {  
 // Check sent messages.  
 for (String recipient : user.getSentMessages().keySet()) {  
 // Only show messages if the sender or receiver is the logged-in user.  
 if (user.equals(*loginUser*) || userDatabase.getUser(recipient).equals(*loginUser*)) {  
 for (Message msg : user.getSentMessages().get(recipient)) {  
 if (msg.getContent().toLowerCase().contains(keyword)) {  
 System.***out***.print("[FROM YOU] "); // Indicate if message was sent by current user.  
 msg.displayMessage();  
 foundMessages = true;  
 }  
 }  
 }  
 }  
 // Check received messages.  
 for (String sender : user.getReceivedMessages().keySet()) {  
 if (user.equals(*loginUser*) || userDatabase.getUser(sender).equals(*loginUser*)) {  
 for (Message msg : user.getReceivedMessages().get(sender)) {  
 if (msg.getContent().toLowerCase().contains(keyword)) {  
 System.***out***.print("[TO YOU] "); // Indicate if message was received by current user.  
 msg.displayMessage();  
 foundMessages = true;  
 }  
 }  
 }  
 }  
 }  
 if (!foundMessages) {  
 // Key Word is the Reciever Name Here  
 System.***out***.println("No messages found containing '" + keyword + "'.");  
  
 }  
 break;  
  
 case 6: // Delete Account  
 System.***out***.println("\n--- Delete Account ---");  
 System.***out***.println("Are you sure you want to delete your account? This action cannot be undone.");  
 System.***out***.print("Type your username ('" + *loginUser*.getUsername() + "') to confirm: ");  
 String confirmUsername = scanner.nextLine();  
  
 if (confirmUsername.equals(*loginUser*.getUsername())) {  
 userDatabase.deleteUser(*loginUser*.getUsername()); // Delete user account.  
 *loginUser* = null; // Log out the user after deletion.  
 } else {  
 System.***out***.println("Confirmation failed. Account not deleted.");  
 }  
 break;  
  
 case 7: // NEW FEATURE: View Recent Logins (Uses LinkedList in UserDatabase)  
 System.***out***.println("\n--- Recent Logins ---");  
 List<String> recentLogins = userDatabase.getRecentLogins();  
 if (recentLogins.isEmpty()) {  
 System.***out***.println("No recent logins to display.");  
 } else {  
 System.***out***.println("Last " + recentLogins.size() + " logins (most recent first):");  
 // Display in reverse order of login (most recent first).  
 for (int i = recentLogins.size() - 1; i >= 0; i--) {  
 System.***out***.println("- " + recentLogins.get(i));  
 }  
 }  
 break;  
  
 case 8: // NEW FEATURE: View Last N Messages (Reverse Order) (Uses Stack in Message)  
 System.***out***.print("Enter username to view messages: ");  
 String chatUserForStack = scanner.nextLine();  
 User targetChatUser = userDatabase.getUser(chatUserForStack);  
  
 if (targetChatUser == null) {  
 System.***out***.println("User with username '" + chatUserForStack + "' not found.");  
 break;  
 }  
  
 System.***out***.print("Enter number of last messages to view: ");  
 int numMessages = *getValidatedInput*(scanner);  
  
 System.***out***.println("\n--- Last " + numMessages + " messages with " + chatUserForStack + " (Newest First) ---");  
 // Call the new method that uses a Stack to get messages in reverse order.  
 ArrayList<Message> lastNMessages = *messageObj*.getMessagesReverseChronological(*loginUser*, targetChatUser, numMessages);  
  
 if (lastNMessages.isEmpty()) {  
 System.***out***.println("No messages to display or less than " + numMessages + " messages with " + chatUserForStack + ".");  
 } else {  
 for (Message message : lastNMessages) {  
 message.displayMessage();  
 }  
 }  
 break;  
  
 case 9: // NEW FEATURE: View All Users (Alphabetical) (Uses Binary Search Tree in UserDatabase)  
 System.***out***.println("\n--- All Registered Users (Alphabetical Order) ---");  
 List<String> allSortedUsers = userDatabase.getAllUsernamesSorted(); // Get usernames from BST.  
 if (allSortedUsers.isEmpty()) {  
 System.***out***.println("No registered users found.");  
 } else {  
 for (String usernameInOrder : allSortedUsers) {  
 User user = userDatabase.getUser(usernameInOrder); // Retrieve full user object for online status.  
 System.***out***.println("- " + user.getUsername() + " (" + (user.isOnline() ? "Online" : "Offline") + ")");  
 }  
 }  
 break;  
  
 case 10: // Logout  
 userDatabase.logout(*loginUser*); // Log out the current user.  
 *loginUser* = null; // Clear the logged-in user.  
 break;  
  
 case 11: // Exit  
 if (*loginUser* != null) {  
 userDatabase.logout(*loginUser*); // Log out before exiting if a user is logged in.  
 }  
 System.***out***.println("Exiting from the Chat App. Goodbye!");  
 scanner.close(); // Close the scanner.  
 return; // Terminate the program.  
  
 default:  
 System.***out***.println("Invalid option. Please try again.");  
 }  
 }  
 }  
 }  
  
 // Helper method to get validated integer input from the user.  
 private static int getValidatedInput(Scanner scanner) {  
 while (true) {  
 try {  
 return Integer.*parseInt*(scanner.nextLine());  
 } catch (NumberFormatException e) {  
 System.***out***.print("Invalid input. Please enter a number: ");  
 }  
 }  
 }  
  
 // Helper method to validate email format using a regular expression.  
 private static boolean isValidEmail(String email) {  
 // Simple regex for basic email validation.  
 String emailRegex = "^[a**-**zA**-**Z0**-**9\_+&\*-]+(?:\\.[a**-**zA**-**Z0**-**9\_+&\*-]+)\*@(?:[a**-**zA**-**Z0**-**9-]+\\.)+[a**-**zA**-**Z]{2,7}$";  
 Pattern pattern = Pattern.*compile*(emailRegex);  
 return email != null && pattern.matcher(email).matches();  
 }  
}

Config

// Core Data Base of the Project  
  
public class Config {  
  
 // Using relative paths makes the application more portable.  
 // If you need specific absolute paths, you can change them here.  
  
 public static final String ***ACCOUNTS\_FILE*** = "accounts.txt";  
  
 public static final String ***MESSAGES\_FILE*** = "Messages.txt";  
  
}

Message

import java.io.\*;  
// For the Setting of the Timing Format  
import java.time.LocalDateTime;  
import java.time.format.DateTimeFormatter;  
import java.util.ArrayList;  
  
/\*  
ArrayList<Message> validMessages → to temporarily store messages read from the file.  
  
ArrayList<String> getChats(User currUser) → to collect usernames you've chatted with.  
  
ArrayList<Message> getMessages(...) → returns list of all messages between two users.  
  
ArrayList<Message> getMessagesReverseChronological(...) → stores reversed messages using Stack.  
 \*/  
import java.util.HashMap;  
// Using the Linked List fOR fETCHING The Data From the FILE and Reading to the File  
import java.util.LinkedList;  
// This is for the Queue Of the Messages In the Order of the FIFO  
import java.util.Queue;  
  
import java.util.Stack; // Imported for the new "View Last N Messages (Reverse Order)" feature.  
  
// The Message class represents a single chat message between two users.  
// It handles message creation, saving to file, loading from file, and various display options.  
public class Message {  
  
 private User sender; // The user who sent the message.  
 private User receiver; // The user who received the message.  
 private String content; // The actual text content of the message.  
 private LocalDateTime timestamp; // The exact date and time the message was sent.  
  
 // Constructor to create a new Message object.  
 public Message(User sender, User receiver, String content) {  
 this.sender = sender;  
 this.receiver = receiver;  
 this.content = content;  
 this.timestamp = LocalDateTime.*now*(); // Automatically sets the current time upon creation.  
 }  
  
 // Saves a single message to the messages file. Appends to the end of the file.  
 public void saveMessageToFile() {  
 // Using try-with-resources ensures the writer is closed automatically.  
  
 try (BufferedWriter writer = new BufferedWriter(new FileWriter(Config.***MESSAGES\_FILE***, true))) { // 'true' means append mode.  
 writer.write("From: " + sender.getUsername());  
 writer.newLine();  
 writer.write("To: " + receiver.getUsername());  
 writer.newLine();  
 writer.write("Message: " + content);  
 writer.newLine();  
 writer.write("Timestamp: " + getFormattedDate() + " " + getFormattedTime());  
 writer.newLine();  
 writer.newLine(); // Add an empty line for readability between messages in the file.  
 } catch (IOException e) {  
 System.***err***.println("Error saving message to file: " + e.getMessage());  
 }  
 }  
  
 // Loads all messages from the messages file into the appropriate user's message queues.  
 // This method is static as it operates on the global set of users.  
 public static void loadDataFromFile(HashMap<String, User> users) {  
 ArrayList<Message> validMessages = new ArrayList<>(); // Temporarily holds messages read from file.  
  
 try (BufferedReader reader = new BufferedReader(new FileReader(Config.***MESSAGES\_FILE***))) {  
 String line;  
 String senderName = null, receiverName = null, messageContent = null, timestampStr = null;  
  
 // Read the file line by line, parsing message details.  
 while ((line = reader.readLine()) != null) {  
 if (line.startsWith("From: ")) {  
 senderName = line.substring(6).trim();  
 } else if (line.startsWith("To: ")) {  
 receiverName = line.substring(4).trim();  
 } else if (line.startsWith("Message: ")) {  
 messageContent = line.substring(9).trim();  
 } else if (line.startsWith("Timestamp: ")) {  
 timestampStr = line.substring(11).trim();  
  
 // Once all parts of a message are read, create a Message object.  
 if (senderName != null && receiverName != null && messageContent != null && timestampStr != null) {  
 DateTimeFormatter formatter = DateTimeFormatter.*ofPattern*("dd-MM-yyyy hh:mm a");  
 LocalDateTime timestamp = LocalDateTime.*parse*(timestampStr, formatter);  
  
 User sender = users.get(senderName); // Retrieve sender User object from map.  
 User receiver = users.get(receiverName); // Retrieve receiver User object from map.  
  
 // Only add message if both sender and receiver users exist in the system.  
 if (sender != null && receiver != null) {  
 Message message = new Message(sender, receiver, messageContent);  
 message.timestamp = timestamp; // Set the actual timestamp from the file.  
 validMessages.add(message);  
 }  
 // Reset temporary variables for the next message.  
 senderName = null;  
 receiverName = null;  
 messageContent = null;  
 timestampStr = null;  
 }  
 }  
 }  
  
 // Clear existing messages for all users before repopulating to prevent duplicates on reload.  
 for (User user : users.values()) {  
 user.getSentMessages().clear();  
 user.getReceivedMessages().clear();  
 }  
  
 // Distribute the loaded messages to the correct user's sent and received message queues.  
 for (Message message : validMessages) {  
 String sName = message.getSender().getUsername();  
 String rName = message.getReceiver().getUsername();  
  
 User senderUser = users.get(sName);  
 User receiverUser = users.get(rName);  
  
 if (senderUser != null && receiverUser != null) {  
 // Add message to sender's 'sent' queue for the recipient.  
 senderUser.getSentMessages().putIfAbsent(rName, new LinkedList<>());  
 senderUser.getSentMessages().get(rName).add(message);  
  
 // Add message to receiver's 'received' queue from the sender.  
 receiverUser.getReceivedMessages().putIfAbsent(sName, new LinkedList<>());  
 receiverUser.getReceivedMessages().get(sName).add(message);  
 }  
 }  
  
 } catch (FileNotFoundException e) {  
 System.***out***.println("Messages file not found. A new one will be created upon sending messages.");  
 } catch (IOException e) {  
 System.***err***.println("Error loading messages from file: " + e.getMessage());  
 }  
 }  
  
 // Rewrites the entire messages file based on the current in-memory message queues.  
 // This is useful after operations like user deletion, to ensure file consistency.  
 public static void rewriteMessagesFile(HashMap<String, User> users) {  
 try (BufferedWriter writer = new BufferedWriter(new FileWriter(Config.***MESSAGES\_FILE***, false))) { // 'false' means overwrite.  
 ArrayList<Message> allMessagesToWrite = new ArrayList<>();  
 // Using a HashMap to ensure each unique message is written only once, even if present in both sent/received queues.  
 HashMap<String, Message> uniqueMessages = new HashMap<>();  
  
 // Collect all messages from all users' sent message queues.  
 for (User user : users.values()) {  
 for (Queue<Message> messagesQueue : user.getSentMessages().values()) {  
 for (Message message : messagesQueue) {  
 // Create a unique key for the message based on its attributes.  
 String messageKey = message.getSender().getUsername() + message.getReceiver().getUsername() +  
 message.getContent() + message.getTimestamp().toString();  
 if (!uniqueMessages.containsKey(messageKey)) {  
 uniqueMessages.put(messageKey, message);  
 allMessagesToWrite.add(message);  
 }  
 }  
 }  
 }  
  
 // Sort all messages chronologically by timestamp before writing to file.  
 // This ensures the file maintains a consistent, ordered history.  
 allMessagesToWrite.sort((m1, m2) -> m1.getTimestamp().compareTo(m2.getTimestamp()));  
  
 // Write each unique, sorted message to the file.  
 for (Message message : allMessagesToWrite) {  
 writer.write("From: " + message.getSender().getUsername());  
 writer.newLine();  
 writer.write("To: " + message.getReceiver().getUsername());  
 writer.newLine();  
 writer.write("Message: " + message.getContent());  
 writer.newLine();  
 writer.write("Timestamp: " + message.getFormattedDate() + " " + message.getFormattedTime());  
 writer.newLine();  
 writer.newLine(); // Add an empty line for separation.  
  
 }  
 } catch (IOException e) {  
 System.***err***.println("Error rewriting messages file: " + e.getMessage());  
 }  
 }  
  
 // --- Getters for Message attributes ---  
 public LocalDateTime getTimestamp()  
  
 {  
 return timestamp;  
 }  
  
 public User getSender() {  
 return sender;  
 }  
  
 public User getReceiver() {  
 return receiver;  
 }  
  
 public String getContent() {  
 return content;  
 }  
  
 // Formats the message timestamp to show time (e.g., "hh:mm AM/PM").  
 public String getFormattedTime() {  
 DateTimeFormatter timeFormatter = DateTimeFormatter.*ofPattern*("hh:mm a");  
 return timestamp.format(timeFormatter);  
 }  
  
 // Formats the message timestamp to show date (e.g., "dd-MM-yyyy").  
 public String getFormattedDate() {  
 DateTimeFormatter dateFormatter = DateTimeFormatter.*ofPattern*("dd-MM-yyyy");  
 return timestamp.format(dateFormatter);  
 }  
  
 // Displays the message content along with sender, receiver, and timestamp to the console.  
 public void displayMessage() {  
 System.***out***.println(sender.getUsername() + " to " + receiver.getUsername() + ": " + content);  
 System.***out***.println("[" + getFormattedTime() + " | " + getFormattedDate() + "]");  
 System.***out***.println();  
 }  
  
 // Sends a message from one user to another. Adds it to both sender's sent queue and receiver's received queue.  
 public void sendMessage(User sender, User receiver, String content) {  
 Message message = new Message(sender, receiver, content);  
  
 // Add message to sender's sent message queue for the specific receiver.  
 sender.getSentMessages().putIfAbsent(receiver.getUsername(), new LinkedList<>());  
 sender.getSentMessages().get(receiver.getUsername()).add(message);  
  
 // Add message to receiver's received message queue from the specific sender.  
 receiver.getReceivedMessages().putIfAbsent(sender.getUsername(), new LinkedList<>());  
 receiver.getReceivedMessages().get(sender.getUsername()).add(message);  
  
 message.displayMessage(); // Display the message immediately after sending.  
 message.saveMessageToFile(); // Persist the message to file.  
 }  
  
 // Retrieves a list of usernames that the current user has chatted with (either sent or received messages).  
 public ArrayList<String> getChats(User currUser) {  
 ArrayList<String> chats = new ArrayList<>();  
  
 // Add all distinct recipients from sent messages.  
 for (String receiverName : currUser.getSentMessages().keySet()) {  
 if (!chats.contains(receiverName)) { // Avoid adding duplicates.  
 chats.add(receiverName);  
 }  
 }  
  
 // Add all distinct senders from received messages.  
 for (String senderName : currUser.getReceivedMessages().keySet()) {  
 if (!chats.contains(senderName)) { // Avoid adding duplicates.  
 chats.add(senderName);  
 }  
 }  
 return chats;  
 }  
  
 // Retrieves all messages between the current user and another specified user, sorted chronologically.  
 public ArrayList<Message> getMessages(User currUser, User user2) {  
 ArrayList<Message> messages = new ArrayList<>();  
  
 // Add messages sent by currUser to user2.  
 if (currUser.getSentMessages().containsKey(user2.getUsername())) {  
 Queue<Message> sentQueue = currUser.getSentMessages().get(user2.getUsername());  
 messages.addAll(sentQueue);  
 }  
  
 // Add messages received by currUser from user2.  
 if (currUser.getReceivedMessages().containsKey(user2.getUsername())) {  
 Queue<Message> receivedQueue = currUser.getReceivedMessages().get(user2.getUsername());  
 messages.addAll(receivedQueue);  
 }  
  
 // Sort all collected messages by their timestamp to ensure chronological order.  
 messages.sort((m1, m2) -> m1.getTimestamp().compareTo(m2.getTimestamp()));  
 return messages;  
 }  
  
 // NEW DSA FEATURE: Retrieves the last 'count' messages between two users in reverse chronological order (newest first).  
 // This demonstrates the use of a Stack (LIFO - Last-In, First-Out).  
 public ArrayList<Message> getMessagesReverseChronological(User currUser, User user2, int count) {  
 ArrayList<Message> allMessages = getMessages(currUser, user2); // First, get all messages in chronological order.  
 Stack<Message> messageStack = new Stack<>(); // Initialize a Stack.  
  
 // Push the most recent 'count' messages onto the stack.  
 // We iterate from the end of the chronologically sorted list backwards.  
 for (int i = allMessages.size() - 1; i >= 0 && count > 0; i--, count--) {  
 messageStack.push(allMessages.get(i));  
 }  
  
 ArrayList<Message> reversedMessages = new ArrayList<>();  
 // Pop messages from the stack. Due to LIFO, they will be in reverse chronological order (newest first).  
 while (!messageStack.isEmpty()) {  
 reversedMessages.add(messageStack.pop());  
 }  
  
 return reversedMessages;  
  
 }  
}

User

import java.util.HashMap;  
import java.util.LinkedList; // Explicitly import LinkedList to highlight its use  
import java.util.Queue;  
  
public class User {  
/\*  
Hash Map named SentMessages may Receiver name is a key And the Data is the Queue of the Messages  
 \*/  
 /\*  
 Hashmap jis ka name ReceivedMessages ha Us may Key ha Sender name ha Aur Data Queue of the Messages Hain  
 \*/  
 private String name;  
 private String password;  
 private String email;  
 private boolean isOnline;  
  
 // Sent messages: HashMap where key is receiver's username, value is a Queue of Messages.  
 // The Queue is implemented using LinkedList, demonstrating FIFO (First-In, First-Out) order for chat history.  
 private HashMap<String, Queue<Message>> sentMessages;  
  
 // Received messages: HashMap where key is sender's username, value is a Queue of Messages.  
 // The Queue is implemented using LinkedList, demonstrating FIFO order for chat history.  
 private HashMap<String, Queue<Message>> receivedMessages;  
  
 public User(String name, String password, String email) {  
 this.name = name;  
 this.password = password;  
 this.email = email;  
 this.isOnline = false; // Initially offline  
 this.sentMessages = new HashMap<>();  
 this.receivedMessages = new HashMap<>();  
 }  
  
 public HashMap<String, Queue<Message>> getSentMessages() {  
  
 return sentMessages;  
 }  
  
 public HashMap<String, Queue<Message>> getReceivedMessages()  
 {  
 return receivedMessages;  
 }  
  
 public String getUsername() {  
 return name;  
 }  
  
 public void setUsername(String username) {  
 this.name = username;  
 }  
  
 public String getPassword() {  
 return password;  
 }  
  
 public void setPassword(String password) {  
 this.password = password;  
 }  
  
 public String getEmail() {  
 return email;  
 }  
  
 public void setEmail(String email) {  
 this.email = email;  
 }  
  
 public boolean isOnline() {  
 return isOnline;  
 }  
  
 public void setOnline(boolean online) {  
 isOnline = online;  
 }  
}

User DataBase

import java.io.\*;  
  
import java.util.HashMap;  
  
import java.util.LinkedList; // Used for recent logins history (LinkedList).  
  
import java.util.List;  
  
import java.util.ArrayList; // For returning lists.  
  
  
// Manages all user accounts, including signup, login, logout, and deletion.  
// It uses a HashMap for quick user lookup, a LinkedList for recent logins, and a BST for sorted usernames.  
public class UserDatabase {  
  
 // HashMap to store all user accounts.  
 // Key: Username (String), Value: User object. Provides O(1) average-case lookup.  
 public HashMap<String, User> userDatabase = new HashMap<>();  
  
  
 // NEW DSA FEATURE: LinkedList to store the usernames of recently logged-in users.  
 // LinkedList is efficient for adding/removing from ends, suitable for a fixed-size history.  
 private LinkedList<String> recentLogins;  
 private static final int ***MAX\_RECENT\_LOGINS*** = 3; // UPDATED: Maximum number of recent logins to track set to 3.  
  
 // NEW DSA FEATURE: Binary Search Tree (BST) to store all usernames in alphabetical order.  
 // Allows for efficient retrieval of all usernames in a sorted manner.  
 private BinarySearchTree usernameBST;  
  
 // Constructor: Initializes data structures and loads existing data from files.  
 public UserDatabase() {  
 recentLogins = new LinkedList<>(); // Initialize the LinkedList for recent logins.  
 usernameBST = new BinarySearchTree(); // Initialize the BST for sorted usernames.  
 loadDataFromFile(); // Load user accounts from file and populate BST.  
  
 Message.*loadDataFromFile*(userDatabase); // Load messages and link them to user objects.  
 }  
  
 // Loads user account data from the accounts file.  
 // Populates the userDatabase HashMap and the usernameBST.  
  
 public void loadDataFromFile() {  
 // Here We Will Use the String Line For the Retrieving of the Data Instead of the Serialization that's Why used the BufferedInput And Buffered outpu  
  
 String line;  
 userDatabase.clear(); // Clear existing users in memory before loading new data.  
 usernameBST.clear(); // Clear the BST to rebuild it with fresh data.  
  
 try (BufferedReader reader = new BufferedReader(new FileReader(Config.***ACCOUNTS\_FILE***))) {  
 String username = null, password = null, email = null;  
 // Read file line by line, parsing user details.  
 while ((line = reader.readLine()) != null) {  
  
 if (line.startsWith("Username:")) {  
 // Holding the Begging Index of the User Naame  
 username = line.substring(9).trim();  
  
 } else if (line.startsWith("Password:")) {  
 password = line.substring(9).trim();  
 } else if (line.startsWith("Email:")) {  
 email = line.substring(6).trim();  
 }  
 // Once all three pieces of information for a user are read, create a User object.  
 if (username != null && password != null && email != null) {  
 User user = new User(username, password, email);  
 userDatabase.put(username, user); // Add user to the HashMap.  
 usernameBST.insert(username); // Add username to the BST for sorted access.  
 // Reset variables for the next user.  
 username = null;  
 password = null;  
 email = null;  
 }  
 }  
 }  
 catch (FileNotFoundException e) {  
 System.***err***.println("Accounts file not found. A new one will be created upon signup.");  
 }  
 catch (IOException e) {  
 System.***err***.println("Error reading accounts file: " + e.getMessage());  
 }  
 }  
  
 // Rewrites the entire accounts file based on the current in-memory userDatabase.  
 // This is crucial after operations like user deletion.  
 public void rewriteAccountsFile() {  
 // 'false' in FileWriter means overwrite the file.  
 try (BufferedWriter writer = new BufferedWriter(new FileWriter(Config.***ACCOUNTS\_FILE***, false))) {  
 // Since Configure Class Holding the Names of the Data Base As the Static That's Why We Used it As the Class Name Pointing Accounts File  
  
 for (User user : userDatabase.values()) { // Iterate through all users in the HashMap.  
 writer.write("Username:" + user.getUsername());  
 writer.newLine();  
 writer.write("Password:" + user.getPassword());  
 writer.newLine();  
 writer.write("Email:" + user.getEmail());  
 writer.newLine();  
 }  
  
 } catch (IOException e) {  
 // this output stream is used to display error messages  
 System.***err***.println("Error rewriting accounts file: " + e.getMessage());  
 }  
 }  
  
 // Handles user signup (registration).  
 public boolean signup(String username, String password, String email) {  
 if (userDatabase.containsKey(username)) { // Check if username already exists.  
 System.***out***.println("Username already taken.");  
 return false;  
 } else {  
 User newUser = new User(username, password, email);  
 userDatabase.put(username, newUser); // Add new user to HashMap.  
 usernameBST.insert(username); // Add new username to BST.  
 addAccountsToFile(username, password, email); // Persist the new account to file.  
 System.***out***.println("User registered successfully!");  
 return true;  
 }  
 }  
  
 // Appends a new user account to the accounts file.  
 public void addAccountsToFile(String username, String password, String email) {  
 // 'true' in FileWriter means append to the file.  
 try (BufferedWriter writer = new BufferedWriter(new FileWriter(Config.***ACCOUNTS\_FILE***, true))) {  
 writer.write("Username:" + username);  
 writer.newLine();  
  
 writer.write("Password:" + password);  
  
 writer.newLine();  
  
 writer.write("Email:" + email);  
  
 writer.newLine();  
  
 } catch (IOException e) {  
 System.***err***.println("Error writing new account to file: " + e.getMessage());  
 }  
 }  
  
 // Handles user login.  
 public User login(String username, String password) {  
 User user = userDatabase.get(username); // Retrieve user from HashMap.  
 if (user != null && user.getPassword().equals(password)) {  
 user.setOnline(true); // Set user status to online.  
  
 // NEW DSA FEATURE (LinkedList): Add user to recent logins.  
 if (recentLogins.contains(username)) {  
 recentLogins.remove(username); // Remove if already present to move it to the end (most recent).  
 }  
 recentLogins.addLast(username); // Add to the end (most recent).  
 // Maintain the fixed size for recent logins.  
 if (recentLogins.size() > ***MAX\_RECENT\_LOGINS***) {  
 recentLogins.removeFirst(); // Remove the oldest login if the limit is exceeded.  
 }  
  
 System.***out***.println("Login successful!");  
 return user;  
 } else {  
 System.***out***.println("Invalid username or password.");  
 return null;  
 }  
 }  
  
 // Logs out a user (sets their online status to false).  
 public void logout(User user) {  
 if (user != null) {  
 user.setOnline(false);  
 System.***out***.println(user.getUsername() + " has been logged out.");  
 }  
 }  
  
 // Deletes a user account from the system.  
 public boolean deleteUser(String username) {  
 if (userDatabase.containsKey(username)) {  
 userDatabase.remove(username); // Remove from in-memory HashMap.  
 usernameBST.delete(username); // Remove from BST.  
  
 rewriteAccountsFile(); // Rewrite accounts file to remove the deleted user.  
  
 // Re-load and rewrite messages to ensure the deleted user's messages are also removed from file.  
 Message.*loadDataFromFile*(userDatabase);  
 Message.*rewriteMessagesFile*(userDatabase);  
  
 // Also remove from recent logins if the deleted user was there.  
 recentLogins.remove(username);  
  
 System.***out***.println("Account '" + username + "' deleted successfully.");  
 return true;  
 } else {  
 System.***out***.println("User '" + username + "' not found.");  
 return false;  
 }  
  
 }  
  
 // NEW DSA FEATURE (LinkedList): Returns a list of recently logged-in usernames.  
 // Returns a new ArrayList to prevent external modification of the internal LinkedList.  
 public List<String> getRecentLogins()  
 {  
  
 return new ArrayList<>(recentLogins);  
  
 }  
  
 // NEW DSA FEATURE (BST): Returns all usernames sorted alphabetically using the BST's inorder traversal.  
 public List<String> getAllUsernamesSorted() {  
 return usernameBST.inorderTraversal();  
 }  
  
 // Retrieves a User object by username from the database HashMap.  
 public User getUser(String username) {  
 return userDatabase.get(username);  
 }  
  
 // Returns the entire user database HashMap.  
 public HashMap<String, User> getUserDatabase() {  
 return userDatabase;  
 }  
}