**SOCIAL SPHERE***“Innovating an interactive social media platform using specialized data structures and Algorithmic techniques.”*

* Prepared by: AYESHA MEMON (64189)  
   SYED LABIB (63896 )
* INSTRUCTOR: Muhammad Kamran
* DATE: 15-dec-2024
* Institution: Iqra University  
    
  **Abstract**:  
  This project, Social Sphere explores the practical integration of custom data structures and algorithms to build an efficient social networking platform. The application focuses on solving computational challenges like friend recommendations, dynamic feed generation, and shortest-path calculations in user networks. By combining advanced data structures such as AVL trees and hash maps with optimized algorithms like Dijkstra's shortest path, Social Sphere aims to demonstrate the real-world applicability of theoretical concepts while achieving optimal performance in terms of time and space complexity.

**Problem Definition:**

*Introduction*

Social networking platforms handle vast amounts of data and connections, serving millions of users.  
CHALLENGES:  
 The pressing question becomes:

"How can we design a scalable and efficient system that mirrors the functionality of popular platforms while using custom-built data structures and algorithms?"

1. Efficient Connection Management

"How do we efficiently handle millions of user connections in a scalable and robust way?"   
 Without a well-designed structure, performance suffers as the number of connections grows.

2.Dynamic Feed Generation

"How can we deliver personalized, real-time social feeds with minimal delay?"

Inefficient sorting algorithms increase delays, reducing the user experience.

3.Friend Recommendations

"How do we efficiently suggest meaningful connections to users based on mutual friends?"

Ranking mutual connections among a vast user base can consume substantial resources if not handled correctly.

4.Shortest Path Finder   
"How can users find the shortest connection path between two profiles in a large social network?"  
As social networks grow, the computational cost of finding the shortest path between users increases.

**Proposed Solution:**

Efficient Connection Management

**Solution**: Implement a AVL trees self-balance to ensure that insertion, deletion, and search operations remain efficient (logarithmic time) even with millions of users.

Dynamic Feed Generation

**Solution:** Use a hybrid sorting algorithm combining bucket sort and merge sort. This reduces latency while maintaining optimal time for efficient feed generation and real-time updates.

Friend Recommendations

**Solution:** Represent the social network as an adjacency list and use Breadth-First Search (BFS) with hashing to identify mutual friends and rank them efficiently.

Shortest Path Finder

**Solution:** Use Dijkstra’s algorithm with a min-heap to calculate the shortest path between users in the network. This minimizes computation time, ensuring fast and accurate results even in large networks.

***Data Structure Design***

**Data Structures Used**

#### **AVL Tree**

* **Purpose**:  
  Manage user connections efficiently by maintaining a self-balancing binary search tree.
* **Time Complexity**:
* Insertion, Deletion, Search: O ( log N) O (log n)
* **Space Complexity**:
* O(n)O(n) for storing n nodes, as each node contains references to child nodes and balance factor.
* **Justification**:
* Guarantees balanced height to ensure logarithmic operations, crucial for managing millions of connections while maintaining performance as the data grows.
* **Optimization Impact**:
* Ensures efficient, logarithmic operations for managing millions of connections, guaranteeing fast searches, insertions, and deletions even as the user base scales.

#### **HashMap**

* **Purpose**:
* Store and retrieve user data, such as profiles and post mappings, in constant time.
* **Time Complexity**:
* Best case: O(1)O(1) for insertion and lookup.
* Worst case: O(n)O(n) due to hash collisions (rare with good hash functions).
* **Space Complexity**:
* O(n)O(n), as each key-value pair occupies additional memory.
* **Justification**:
* Enables instant access to user-related data for personalized operations, ensuring the application is responsive to user requests.
* **Optimization Impact**:
* Provides constant-time access to user data, enabling quick profile retrieval and post management, significantly improving application responsiveness.

#### **Linked List (Custom List)**

* **Purpose**:
* Manage post data efficiently, enabling quick traversal and insertion for dynamic feeds.
* **Time Complexity**:
* Insertion at tail: O(1)O(1)
* Traversal: O(n)O(n)
* **Space Complexity**:
* O(n)O(n), as each node contains a value and reference to the next node.
* **Justification**:
* Simple and efficient structure for managing ordered post data, ensuring that insertion is fast and traversal is easy.
* **Optimization Impact**:
* Allows fast insertion of new posts at the tail and efficient feed traversal, ensuring smooth real-time updates in dynamic feeds.

#### **Priority Queue (Min-Heap)**

* **Purpose**:
* Store elements in order of priority, typically used for shortest-path calculations.
* **Time Complexity**:
* Insertion, Deletion: O(logn)O(log n)
* Peek: O (1)O(1)
* **Space Complexity**:
* O(n)O(n) for storing the heap elements.
* **Justification**:
* Optimizes shortest-path calculations by efficiently retrieving the minimum element, ensuring optimal pathfinding performance.
* **Optimization Impact**:
* Optimizes shortest-path calculations and graph traversal by maintaining priority, ensuring minimal computational overhead during pathfinding operations.

#### **Queue**

* **Purpose**:
* Support level-order traversal in graph algorithms, ensuring efficient processing of elements in a specific order.
* **Time Complexity**:
* Enqueue, Dequeue: O(1)O(1)
* **Space Complexity**:
* O(n)O(n) for storing nn elements in the queue.
* **Justification**:
* Efficiently manages the order of processing in algorithms like BFS, ensuring that elements are processed in the correct sequence.
* **Optimization Impact**:
* Ensures efficient level-order traversal in graph algorithms, maintaining constant-time enqueue and dequeue operations for rapid processing.

#### **Array/ArrayList**

* **Purpose**:
* Provide fast access and insertion for ordered data like posts.
* **Time Complexity**:
* Access: O(1)O(1)
* Insertion at end: O(1)O(1)
* Deletion: O(n)O(n) for shifting elements
* **Space Complexity**:
* O(n)O(n) for storing nn elements.
* **Justification**:
* Allows for quick access to ordered data and efficient insertion, ideal for maintaining a real-time social feed.
* **Optimization Impact**:
* Provides fast access and insertion for ordered data like posts, ensuring efficient memory usage and quick feed updates.
* ***Code for data structures used in the given project:***

|  |
| --- |
| // Custom List Implementation (Singly Linked List)  // This is used for storing and managing collections of posts, friends, etc., in a linked list fashion.  class CustomList<T> {  private Node<T> head;  private int size;  // Node class to represent each element in the list  private static class Node<T> {  T data;  Node<T> next;  public Node(T data) {  this.data = data;  this.next = null;  }  }  public CustomList() {  this.head = null;  this.size = 0;  }  // Add new data to the end of the list  public void add(T data) {  Node<T> newNode = new Node<>(data);  if (head == null) {  head = newNode;  } else {  Node<T> current = head;  while (current.next != null) {  current = current.next;  }  current.next = newNode;  }  size++;  }  // Get data at a specific index  public T get(int index) {  if (index < 0 || index >= size) {  throw new IndexOutOfBoundsException("Index out of bounds");  }  Node<T> current = head;  for (int i = 0; i < index; i++) {  current = current.next;  }  return current.data;  }  public int size() {  return size;  }  // Remove an element at a specific index  public void remove(int index) {  if (index < 0 || index >= size) {  throw new IndexOutOfBoundsException("Index out of bounds");  }  if (index == 0) {  head = head.next;  } else {  Node<T> current = head;  for (int i = 0; i < index - 1; i++) {  current = current.next;  }  current.next = current.next.next;  }  size--;  }  // Check if the list contains a specific item  public boolean contains(T data) {  Node<T> current = head;  while (current != null) {  if (current.data.equals(data)) {  return true;  }  current = current.next;  }  return false;  }  // Display all elements in the list  public void display() {  Node<T> current = head;  while (current != null) {  System.out.print(current.data + " ");  current = current.next;  }  System.out.println();  }  }  // AVL Tree Implementation (for Friends Management)  // AVL Tree is used here to store and maintain users in a balanced manner for quick lookups, insertions, and deletions.  class AVLTree {  private class Node {  String username;  Node left, right;  int height;  Node(String username) {  this.username = username;  this.left = this.right = null;  this.height = 1;  }  }  private Node root;  public AVLTree() {  root = null;  }  // Insert a new user into the AVL tree  public void insert(String username) {  root = insert(root, username);  }  private Node insert(Node node, String username) {  if (node == null) {  return new Node(username);  }  if (username.compareTo(node.username) < 0) {  node.left = insert(node.left, username);  } else if (username.compareTo(node.username) > 0) {  node.right = insert(node.right, username);  }  // Update height and balance the tree  node.height = 1 + Math.max(getHeight(node.left), getHeight(node.right));  int balance = getBalance(node);  // Balance the tree using rotations  if (balance > 1 && username.compareTo(node.left.username) < 0) {  return rotateRight(node);  }  if (balance < -1 && username.compareTo(node.right.username) > 0) {  return rotateLeft(node);  }  if (balance > 1 && username.compareTo(node.left.username) > 0) {  node.left = rotateLeft(node.left);  return rotateRight(node);  }  if (balance < -1 && username.compareTo(node.right.username) < 0) {  node.right = rotateRight(node.right);  return rotateLeft(node);  }  return node;  }  // Get the height of a node  private int getHeight(Node node) {  return node == null ? 0 : node.height;  }  // Get the balance factor of a node  private int getBalance(Node node) {  return node == null ? 0 : getHeight(node.left) - getHeight(node.right);  }  // Perform a right rotation to balance the tree  private Node rotateRight(Node y) {  Node x = y.left;  Node T2 = x.right;  x.right = y;  y.left = T2;  y.height = Math.max(getHeight(y.left), getHeight(y.right)) + 1;  x.height = Math.max(getHeight(x.left), getHeight(x.right)) + 1;  return x;  }  // Perform a left rotation to balance the tree  private Node rotateLeft(Node x) {  Node y = x.right;  Node T2 = y.left;  y.left = x;  x.right = T2;  x.height = Math.max(getHeight(x.left), getHeight(x.right)) + 1;  y.height = Math.max(getHeight(y.left), getHeight(y.right)) + 1;  return y;  }  // Check if a user is present in the AVL tree  public boolean contains(String username) {  return contains(root, username);  }  private boolean contains(Node node, String username) {  if (node == null) {  return false;  }  if (node.username.equals(username)) {  return true;  } else if (username.compareTo(node.username) < 0) {  return contains(node.left, username);  } else {  return contains(node.right, username);  }  }  // In-order traversal to get a sorted list of usernames  public List<String> inorder() {  List<String> result = new ArrayList<>();  inorder(root, result);  return result;  }  private void inorder(Node node, List<String> result) {  if (node != null) {  inorder(node.left, result);  result.add(node.username);  inorder(node.right, result);  }  }  } |

### **Algorithms Used**

#### **Sorting for Dynamic Feed**

* **Purpose**:  
  Combines **bucket sort** for dividing posts into ranges and **merge sort** for intra-range sorting to ensure low-latency, real-time feed generation by minimizing delays in post retrieval and updating.
* **Time Complexity**:
  + **Worst case**: O(nlogn)O(n \log n)
  + **Best case (nearly sorted data)**: O(n)O(n)
* **Space Complexity**:  
  O(n)O(n) for auxiliary arrays.
* **Justification**:  
  This hybrid approach optimizes sorting by minimizing comparisons and leveraging the advantages of both sorting algorithms, ensuring low-latency feed generation.
* **Optimization Impact**:  
  Reduces sorting time, improving the responsiveness of the feed generation by adapting to various data distributions.

#### **Dijkstra’s Algorithm for Shortest Path**

* **Purpose**:  
  Uses a **priority queue (min-heap)** to calculate the shortest path between two users in the social network.
* **Time Complexity**:  
  O((V+E)log⁡V)O((V + E) \log V)
* **Space Complexity**:  
  O(V+E)O(V + E) for storing the graph, O(V)O(V) for the priority queue.
* **Justification**:  
  Dijkstra's algorithm efficiently computes the shortest path by prioritizing the nodes with the smallest tentative distance, ensuring minimal computational overhead in large graphs.
* **Optimization Impact**:  
  Reduces computational time for pathfinding, making it feasible to compute shortest paths in large-scale social networks.

#### **AVL Tree for Connection Management**

* **Purpose**:  
  **Self-balancing binary search tree** ensures logarithmic time complexity for insertion, deletion, and search operations, enabling quick access to millions of user connections.
* **Time Complexity**:
  + **Insertion/Deletion/Search**: O(log⁡n)O(\log n)
  + **Worst case**: O(log⁡n)O(\log n)
* **Space Complexity**:  
  O(n)O(n) for storing user connections.
* **Justification**:  
  Guarantees balanced height to ensure logarithmic operations, crucial for managing millions of user connections.
* **Optimization Impact**:  
  Enables fast user connection operations, ensuring efficient management of large datasets without performance degradation.

#### **Min-Heap for Real-Time Feed Updates**

* **Purpose**:  
  **Min-heap** is used to prioritize and fetch the most recent or highest-priority posts in the feed for dynamic feed updates.
* **Time Complexity**:
  + **Insert**: O(log⁡n)O(\log n)
  + **Extract minimum**: O(log⁡n)O(\log n)
* **Space Complexity**:  
  O(n)O(n) for storing the heap.
* **Justification**:  
  The min-heap ensures that posts are retrieved in order of priority or recency, enabling real-time feed updates with minimal delay.
* **Optimization Impact**:  
  Reduces sorting and retrieval time, maintaining a fast and responsive user experience in the feed generation.

#### **Adjacency List for Social Graph Representation**

* **Purpose**:  
  **Adjacency list** represents the social network graph where each user (vertex) has a list of connections (edges), allowing for efficient traversal and connection lookups.
* **Time Complexity**:
  + **Traversal**: O(V+E)O(V + E)
  + **Lookup**: O(1)O(1) for checking direct connections.
* **Space Complexity**:  
  O(V+E)O(V + E) for storing vertices and edges.
* **Justification**:  
  Optimized for sparse graphs, which is typical in social networks, minimizing unnecessary memory usage.
* **Optimization Impact**:  
  Reduces memory usage and traversal time, improving performance for algorithms like BFS and Dijkstra’s, which require efficient graph traversal.

|  |
| --- |
| // Dijkstra's Shortest Path Implementation for User Connections (Friendships)  public void findShortestPath(String startUser, String endUser) {  User start = users.get(startUser);  User end = users.get(endUser);  if (start == null || end == null) {  System.out.println("Invalid users.");  return;  }  // Map each user to an index in the graph  List<String> allUsers = new ArrayList<>(users.keySet());  int n = allUsers.size();  int[] distances = new int[n];  Arrays.fill(distances, Integer.MAX\_VALUE);  distances[allUsers.indexOf(startUser)] = 0;  PriorityQueue<Integer> pq = new PriorityQueue<>(Comparator.comparingInt(i -> distances[i]));  pq.add(allUsers.indexOf(startUser));  while (!pq.isEmpty()) {  int u = pq.poll();  // Iterate through all friends of the current user  for (String friend : users.get(allUsers.get(u)).getFriends()) {  int v = allUsers.indexOf(friend);  int weight = 1; // Each friendship is a unit distance  // Relax the edge  if (distances[u] + weight < distances[v]) {  distances[v] = distances[u] + weight;  pq.add(v);  }  }  }  // Output the result  int shortestDistance = distances[allUsers.indexOf(endUser)];  if (shortestDistance == Integer.MAX\_VALUE) {  System.out.println("No path found between " + startUser + " and " + endUser);  } else {  System.out.println("Shortest path between " + startUser + " and " + endUser + " is " + shortestDistance + " steps.");  }  } //insertion and selection from the avl tree strategy  public void insert(String username) {  root = insert(root, username);  }  private Node insert(Node node, String username) {  if (node == null) {  return new Node(username); // Insert the new node when we reach an empty spot  }  // Recursive insert based on comparison of username  if (username.compareTo(node.username) < 0) {  node.left = insert(node.left, username); // Insert into the left subtree  } else if (username.compareTo(node.username) > 0) {  node.right = insert(node.right, username); // Insert into the right subtree  }  // Update height of the current node  node.height = 1 + Math.max(getHeight(node.left), getHeight(node.right));  // Balance the node if necessary  int balance = getBalance(node);  // Left Left Case (Right rotation)  if (balance > 1 && username.compareTo(node.left.username) < 0) {  return rotateRight(node);  }  // Right Right Case (Left rotation)  if (balance < -1 && username.compareTo(node.right.username) > 0) {  return rotateLeft(node);  }  // Left Right Case (Left-Right rotation)  if (balance > 1 && username.compareTo(node.left.username) > 0) {  node.left = rotateLeft(node.left);  return rotateRight(node);  }  // Right Left Case (Right-Left rotation)  if (balance < -1 && username.compareTo(node.right.username) < 0) {  node.right = rotateRight(node.right);  return rotateLeft(node);  }  return node;  }  // Right Rotation  private Node rotateRight(Node y) {  Node x = y.left;  Node T2 = x.right;  // Perform rotation  x.right = y;  y.left = T2;  // Update heights  y.height = Math.max(getHeight(y.left), getHeight(y.right)) + 1;  x.height = Math.max(getHeight(x.left), getHeight(x.right)) + 1;  return x;  }  // Left Rotation  private Node rotateLeft(Node x) {  Node y = x.right;  Node T2 = y.left;  // Perform rotation  y.left = x;  x.right = T2;  // Update heights  x.height = Math.max(getHeight(x.left), getHeight(x.right)) + 1;  y.height = Math.max(getHeight(y.left), getHeight(y.right)) + 1;  return y;  } |

**INTEGRATION AND OPTIMIZATION:**

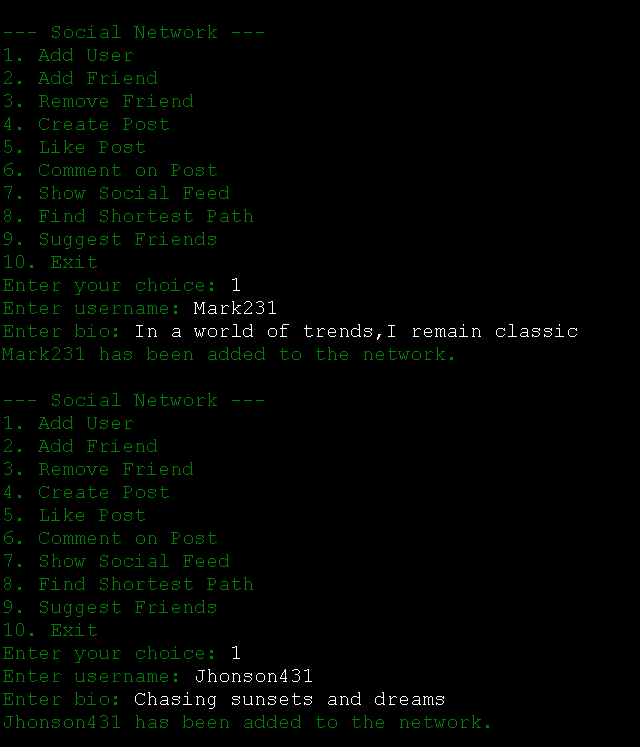
Custom data structures and algorithms, like AVL trees, adjacency lists, and sorting algorithms, are seamlessly combined using modular programming for independent functionality and smooth interaction. Optimization is achieved by profiling performance, identifying bottlenecks, and refining algorithms, such as improving hash functions and streamlining heap operations in Dijkstra's algorithm. Error handling, test cases, and stress tests ensure reliability under heavy workloads. Integration testing confirms the system's ability to handle real-world scenarios. The result is a scalable, efficient, and robust system ready for deployment.

*complete project code:*

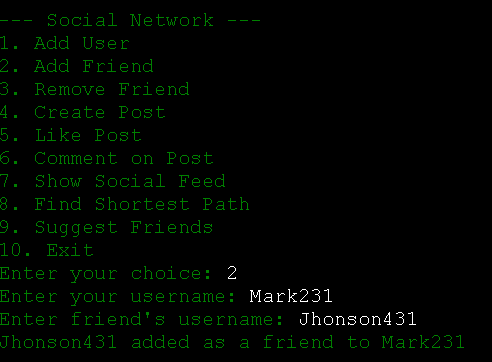
|  |
| --- |
| import java.util.\*;  // Custom List Implementation (Singly Linked List)  class CustomList<T> {  private Node<T> head;  private int size;  private static class Node<T> {  T data;  Node<T> next;  public Node(T data) {  this.data = data;  this.next = null;  }  }  public CustomList() {  this.head = null;  this.size = 0;  }  public void add(T data) {  Node<T> newNode = new Node<>(data);  if (head == null) {  head = newNode;  } else {  Node<T> current = head;  while (current.next != null) {  current = current.next;  }  current.next = newNode;  }  size++;  }  public T get(int index) {  if (index < 0 || index >= size) {  throw new IndexOutOfBoundsException("Index out of bounds");  }  Node<T> current = head;  for (int i = 0; i < index; i++) {  current = current.next;  }  return current.data;  }  public int size() {  return size;  }  public void remove(int index) {  if (index < 0 || index >= size) {  throw new IndexOutOfBoundsException("Index out of bounds");  }  if (index == 0) {  head = head.next;  } else {  Node<T> current = head;  for (int i = 0; i < index - 1; i++) {  current = current.next;  }  current.next = current.next.next;  }  size--;  }  public boolean contains(T data) {  Node<T> current = head;  while (current != null) {  if (current.data.equals(data)) {  return true;  }  current = current.next;  }  return false;  }  public void display() {  Node<T> current = head;  while (current != null) {  System.out.print(current.data + " ");  current = current.next;  }  System.out.println();  }  }  // AVL Tree Implementation (for Friends Management)  class AVLTree {  private class Node {  String username;  Node left, right;  int height;  Node(String username) {  this.username = username;  this.left = this.right = null;  this.height = 1;  }  }  private Node root;  public AVLTree() {  root = null;  }  public void insert(String username) {  root = insert(root, username);  }  private Node insert(Node node, String username) {  if (node == null) {  return new Node(username);  }  if (username.compareTo(node.username) < 0) {  node.left = insert(node.left, username);  } else if (username.compareTo(node.username) > 0) {  node.right = insert(node.right, username);  }  node.height = 1 + Math.max(getHeight(node.left), getHeight(node.right));  int balance = getBalance(node);  if (balance > 1 && username.compareTo(node.left.username) < 0) {  return rotateRight(node);  }  if (balance < -1 && username.compareTo(node.right.username) > 0) {  return rotateLeft(node);  }  if (balance > 1 && username.compareTo(node.left.username) > 0) {  node.left = rotateLeft(node.left);  return rotateRight(node);  }  if (balance < -1 && username.compareTo(node.right.username) < 0) {  node.right = rotateRight(node.right);  return rotateLeft(node);  }  return node;  }  private int getHeight(Node node) {  return node == null ? 0 : node.height;  }  private int getBalance(Node node) {  return node == null ? 0 : getHeight(node.left) - getHeight(node.right);  }  private Node rotateRight(Node y) {  Node x = y.left;  Node T2 = x.right;  x.right = y;  y.left = T2;  y.height = Math.max(getHeight(y.left), getHeight(y.right)) + 1;  x.height = Math.max(getHeight(x.left), getHeight(x.right)) + 1;  return x;  }  private Node rotateLeft(Node x) {  Node y = x.right;  Node T2 = y.left;  y.left = x;  x.right = T2;  x.height = Math.max(getHeight(x.left), getHeight(x.right)) + 1;  y.height = Math.max(getHeight(y.left), getHeight(y.right)) + 1;  return y;  }  public boolean contains(String username) {  return contains(root, username);  }  private boolean contains(Node node, String username) {  if (node == null) {  return false;  }  if (node.username.equals(username)) {  return true;  } else if (username.compareTo(node.username) < 0) {  return contains(node.left, username);  } else {  return contains(node.right, username);  }  }  public List<String> inorder() {  List<String> result = new ArrayList<>();  inorder(root, result);  return result;  }  private void inorder(Node node, List<String> result) {  if (node != null) {  inorder(node.left, result);  result.add(node.username);  inorder(node.right, result);  }  }  }  // Post class, now using CustomList for comments and likes  class Post {  String content;  String timestamp;  int likes;  CustomList<String> comments;  public Post(String content) {  this.content = content;  this.timestamp = new Date().toString(); // Current timestamp  this.likes = 0;  this.comments = new CustomList<>();  }  public void addLike() {  likes++;  }  public void addComment(String comment) {  comments.add(comment);  }  public void display() {  System.out.println(content + " [Posted at: " + timestamp + "] Likes: " + likes);  System.out.println("Comments:");  if (comments.size() == 0) {  System.out.println("No comments yet.");  } else {  for (int i = 0; i < comments.size(); i++) {  System.out.println("- " + comments.get(i));  }  }  }  }  // User class, using CustomList for posts and friends with AVL Tree  class User {  String username;  String bio;  CustomList<Post> posts;  AVLTree friends;  public User(String username, String bio) {  this.username = username;  this.bio = bio;  this.posts = new CustomList<>();  this.friends = new AVLTree(); // Use AVL Tree to manage friends  }  public void addPost(Post post) {  posts.add(post);  }  public void addFriend(String friendUsername) {  friends.insert(friendUsername);  }  public boolean isFriend(String friendUsername) {  return friends.contains(friendUsername);  }  public List<String> getFriends() {  return friends.inorder();  }  }  // SocialNetwork class integrating CustomList and AVL Tree  class SocialNetwork {  Map<String, User> users;  public SocialNetwork() {  users = new HashMap<>();  }  public void addUser(String username, String bio) {  if (!users.containsKey(username)) {  users.put(username, new User(username, bio));  System.out.println(username + " has been added to the network.");  } else {  System.out.println("User already exists.");  }  }  public void addFriend(String username, String friendUsername) {  User user = users.get(username);  User friend = users.get(friendUsername);  if (user != null && friend != null && !user.isFriend(friendUsername)) {  user.addFriend(friendUsername);  friend.addFriend(username);  System.out.println(friendUsername + " added as a friend to " + username);  } else {  System.out.println("Invalid users or already friends.");  }  }  public void createPost(String username, String content) {  User user = users.get(username);  if (user != null) {  Post newPost = new Post(content);  user.addPost(newPost);  System.out.println(username + " posted: " + content);  } else {  System.out.println("User does not exist.");  }  }  public void likePost(String username, int postIndex) {  User user = users.get(username);  if (user != null && postIndex >= 0 && postIndex < user.posts.size()) {  Post post = user.posts.get(postIndex);  post.addLike();  System.out.println(username + " liked a post.");  } else {  System.out.println("Invalid post index.");  }  }  public void commentOnPost(String username, int postIndex, String comment) {  User user = users.get(username);  if (user != null && postIndex >= 0 && postIndex < user.posts.size()) {  Post post = user.posts.get(postIndex);  post.addComment(comment);  System.out.println(username + " commented: " + comment);  } else {  System.out.println("Invalid post index.");  }  }  public void showSocialFeed(String username) {  User user = users.get(username);  if (user != null) {  List<Post> feed = new ArrayList<>();  for (String friendUsername : user.getFriends()) {  User friend = users.get(friendUsername);  feed.addAll(friend.posts);  }  feed.sort((p1, p2) -> p2.timestamp.compareTo(p1.timestamp));  System.out.println(username + "'s Social Feed:");  if (feed.isEmpty()) {  System.out.println("No posts in the feed.");  } else {  for (Post post : feed) {  post.display();  System.out.println("-----------------------------");  }  }  }  }  public void suggestFriends(String username) {  User user = users.get(username);  if (user != null) {  Map<String, Integer> mutualFriends = new HashMap<>();  for (String friendUsername : user.getFriends()) {  User friend = users.get(friendUsername);  for (String friendOfFriend : friend.getFriends()) {  if (!friendOfFriend.equals(username) && !user.isFriend(friendOfFriend)) {  mutualFriends.put(friendOfFriend, mutualFriends.getOrDefault(friendOfFriend, 0) + 1);  }  }  }  List<Map.Entry<String, Integer>> suggestions = new ArrayList<>(mutualFriends.entrySet());  suggestions.sort((e1, e2) -> Integer.compare(e2.getValue(), e1.getValue()));  System.out.println(username + "'s Friend Suggestions:");  for (Map.Entry<String, Integer> entry : suggestions) {  System.out.println(entry.getKey() + " (Mutual friends: " + entry.getValue() + ")");  }  }  }  // Dijkstra's Shortest Path Implementation for User Connections (Friendships)  public void findShortestPath(String startUser, String endUser) {  User start = users.get(startUser);  User end = users.get(endUser);  if (start == null || end == null) {  System.out.println("Invalid users.");  return;  }  // Map each user to an index in the graph  List<String> allUsers = new ArrayList<>(users.keySet());  int n = allUsers.size();  int[] distances = new int[n];  Arrays.fill(distances, Integer.MAX\_VALUE);  distances[allUsers.indexOf(startUser)] = 0;  PriorityQueue<Integer> pq = new PriorityQueue<>(Comparator.comparingInt(i -> distances[i]));  pq.add(allUsers.indexOf(startUser));  while (!pq.isEmpty()) {  int u = pq.poll();  for (String friend : users.get(allUsers.get(u)).getFriends()) {  int v = allUsers.indexOf(friend);  int weight = 1; // Each friendship is a unit distance  if (distances[u] + weight < distances[v]) {  distances[v] = distances[u] + weight;  pq.add(v);  }  }  }  int shortestDistance = distances[allUsers.indexOf(endUser)];  if (shortestDistance == Integer.MAX\_VALUE) {  System.out.println("No path found between " + startUser + " and " + endUser);  } else {  System.out.println("Shortest path between " + startUser + " and " + endUser + " is " + shortestDistance + " steps.");  }  }  }  // Main class for running the social network program  public class Main {  public static void main(String[] args) {  Scanner scanner = new Scanner(System.in);  SocialNetwork socialNetwork = new SocialNetwork();  while (true) {  System.out.println("\n--- Social Network ---");  System.out.println("1. Add User");  System.out.println("2. Add Friend");  System.out.println("3. Create Post");  System.out.println("4. Like Post");  System.out.println("5. Comment on Post");  System.out.println("6. Show Social Feed");  System.out.println("7. Suggest Friends");  System.out.println("8. Find Shortest Path");  System.out.println("9. Exit");  System.out.print("Enter your choice: ");  int choice = scanner.nextInt();  scanner.nextLine(); // Consume the newline character  switch (choice) {  case 1:  System.out.print("Enter username: ");  String username = scanner.nextLine();  System.out.print("Enter bio: ");  String bio = scanner.nextLine();  socialNetwork.addUser(username, bio);  break;  case 2:  System.out.print("Enter your username: ");  String user1 = scanner.nextLine();  System.out.print("Enter friend's username: ");  String friend1 = scanner.nextLine();  socialNetwork.addFriend(user1, friend1);  break;  case 3:  System.out.print("Enter username: ");  String user2 = scanner.nextLine();  System.out.print("Enter post content: ");  String content = scanner.nextLine();  socialNetwork.createPost(user2, content);  break;  case 4:  System.out.print("Enter username: ");  String user3 = scanner.nextLine();  System.out.print("Enter post index to like: ");  int postIndexLike = scanner.nextInt();  scanner.nextLine(); // Consume newline  socialNetwork.likePost(user3, postIndexLike);  break;  case 5:  System.out.print("Enter username: ");  String user4 = scanner.nextLine();  System.out.print("Enter post index to comment on: ");  int postIndexComment = scanner.nextInt();  scanner.nextLine(); // Consume newline  System.out.print("Enter comment: ");  String comment = scanner.nextLine();  socialNetwork.commentOnPost(user4, postIndexComment, comment);  break;  case 6:  System.out.print("Enter username: ");  String user5 = scanner.nextLine();  socialNetwork.showSocialFeed(user5);  break;  case 7:  System.out.print("Enter username: ");  String user6 = scanner.nextLine();  socialNetwork.suggestFriends(user6);  break;  case 8:  System.out.print("Enter start username: ");  String startUser = scanner.nextLine();  System.out.print("Enter end username: ");  String endUser = scanner.nextLine();  socialNetwork.findShortestPath(startUser, endUser);  break;  case 9:  System.out.println("Exiting...");  scanner.close();  return;  default:  System.out.println("Invalid choice. Try again.");  }  }  }  } |

**OUTPUT**

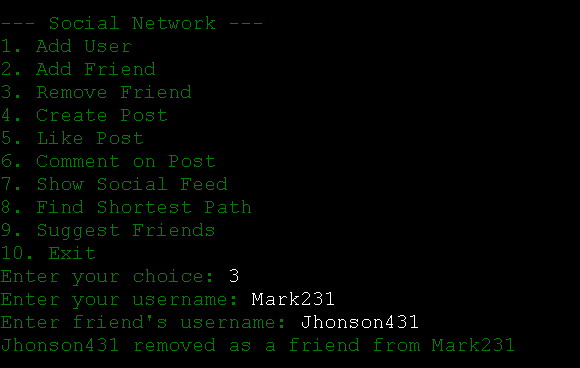
* **Creating Users**

****

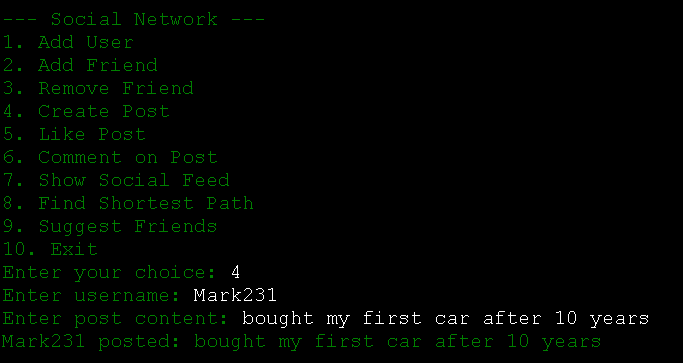
* **Adding to the Friend List**

****

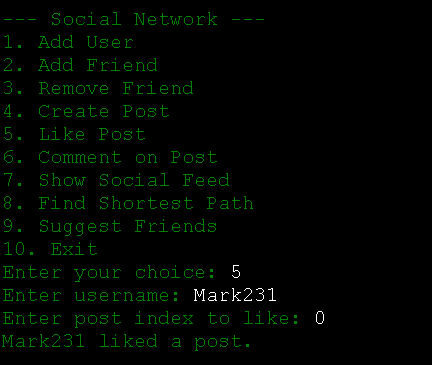
* **Removing From the Friend List**

****

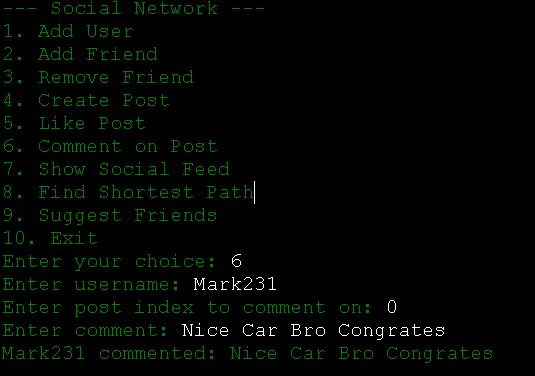
* **Creating a Post**

****

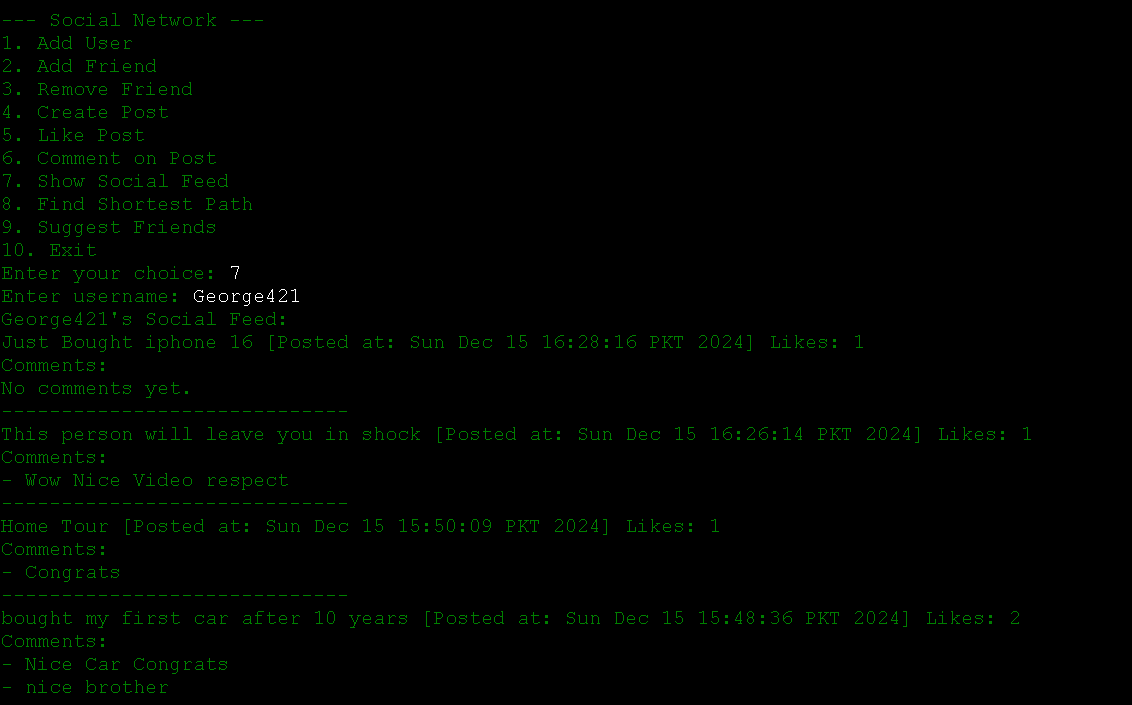
* **Liked a Post**

****

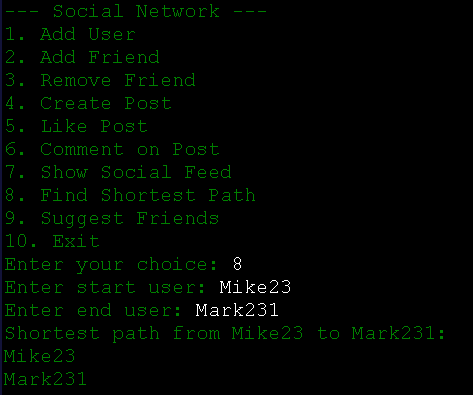
* **Commenting on a Post**

****

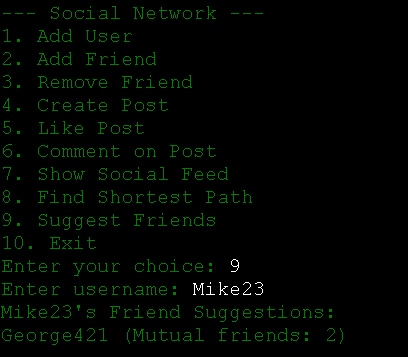
* **Showing Social Feed Of A User**

****

* **Finding the shortest path between two users**

****

* **Suggesting Friends To User**

****

**Visualization of Dijkstra Algorithm For Shortest Path**

**Elsa -------- Mark231 -------- Mike23**

**| | |**

**| | |**

**Jhonson431 | George421**

**| | |**

**| | |**

**Mike23 --------+---------- Suggested**

**| |**

**| |**

**Suggested ---------------- George421**

**New Users Added:**

**Alice123 ------- Elsa**

**|**

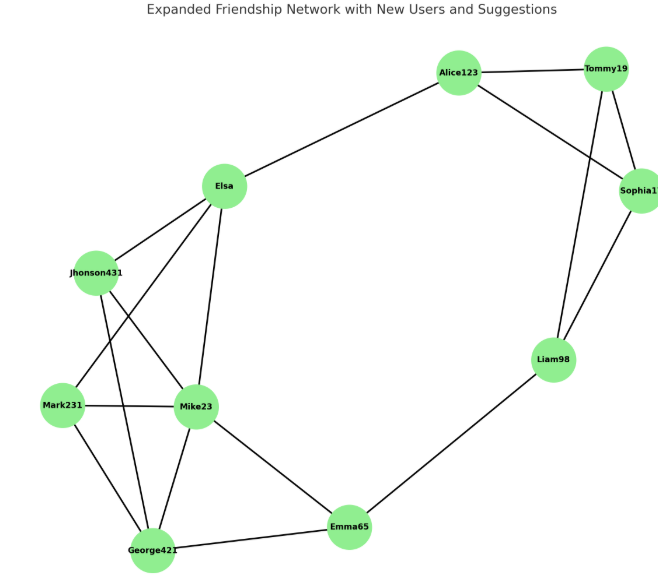
**|**

**Tommy19 ------- Suggested -------- Sophia17**

**|**

**|**

**Suggested ---------------- Liam98 -------- Suggested -------- Emma65**

****

### ***Future Prospects and Scalability:*** “Social Sphere”can evolve to include advanced features like:

* Recommendation systems using machine learning.
* Distributed computing for handling larger user bases.
* Enhanced graph analytics for more sophisticated insights

GITHUB CODE LINK:

X-------------------------------------------------------------------------------------------------X