Atlassian Java Coding & Design Round Interview Preparation

Problem 1: Employee Directory - Closest Common Group

Problem Analysis

Find the closest common parent group for a set of employees in an organizational hierarchy using Java.

Solution Approach

Use tree data structure with LCA (Lowest Common Ancestor) algorithm implemented in Java.

java

```java
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.locks.ReentrantReadWriteLock;
import java.util.stream.Collectors;

class Group {
    private String name;
    private Group parent;
    private Set<Group> children;
    private Set<String> employees;

    public Group(String name) {
        this.name = name;
        this.children = ConcurrentHashMap.newKeySet();
        this.employees = ConcurrentHashMap.newKeySet();
    }

    // Getters and setters
    public String getName() { return name; }
    public Group getParent() { return parent; }
    public void setParent(Group parent) { this.parent = parent; }
    public Set<Group> getChildren() { return children; }
    public Set<String> getEmployees() { return employees; }

    public void addChild(Group child) {
        children.add(child);
        child.setParent(this);
    }

    public void addEmployee(String employee) {
        employees.add(employee);
    }
}

class Employee {
    private String name;
    private Set<Group> groups;

    public Employee(String name, Group group) {
        this.name = name;
        this.groups = ConcurrentHashMap.newKeySet();
        this.groups.add(group);
        group.addEmployee(name);
    }

    public String getName() { return name; }
    public Set<Group> getGroups() { return groups; }
```

```
    public void addToGroup(Group group) {
        groups.add(group);
        group.addEmployee(name);
    }
}

public class EmployeeDirectory {
    private final Map<String, Group> groups;
    private final Map<String, Employee> employees;
    private final ReentrantReadWriteLock lock;

    public EmployeeDirectory() {
        this.groups = new ConcurrentHashMap<>();
        this.employees = new ConcurrentHashMap<>();
        this.lock = new ReentrantReadWriteLock();
    }

    public void addGroup(String name, String parentName) {
        lock.writeLock().lock();
        try {
            Group group = new Group(name);
            groups.put(name, group);

            if (parentName != null && groups.containsKey(parentName)) {
                Group parent = groups.get(parentName);
                parent.addChild(group);
            }
        } finally {
            lock.writeLock().unlock();
        }
    }

    public void addEmployee(String empName, String groupName) {
        lock.writeLock().lock();
        try {
            if (!groups.containsKey(groupName)) {
                return;
            }

            Group group = groups.get(groupName);
            if (employees.containsKey(empName)) {
                employees.get(empName).addToGroup(group);
            } else {
                employees.put(empName, new Employee(empName, group));
            }
        } finally {
            lock.writeLock().unlock();
        }
    }
```

```
}

private List<Group> getPathToRoot(Group group) {
    List<Group> path = new ArrayList<>();
    Group current = group;

    while (current != null) {
        path.add(current);
        current = current.getParent();
    }

    return path;
}

private Group findLCA(List<Group> groups) {
    if (groups.isEmpty()) {
        return null;
    }

    // Get all paths to root
    List<List<Group>> paths = groups.stream()
        .map(this::getPathToRoot)
        .collect(Collectors.toList());

    int minLength = paths.stream()
        .mapToInt(List::size)
        .min()
        .orElse(0);

    // Find LCA by comparing from root down
    for (int i = 0; i < minLength; i++) {
        Group currentGroup = paths.get(0).get(paths.get(0).size() - 1 - i);
        boolean allMatch = paths.stream()
            .allMatch(path -> path.get(path.size() - 1 - i).equals(currentGroup));

        if (!allMatch) {
            return i > 0 ? paths.get(0).get(paths.get(0).size() - i) : null;
        }
    }

    return minLength > 0 ? paths.get(0).get(paths.get(0).size() - minLength) : null;
}

public String getCommonGroupForEmployees(List<String> employeeNames) {
    lock.readLock().lock();
    try {
        Set<Group> allGroups = new HashSet<>();

        // Collect all groups for all employees
        for (String empName : employeeNames) {
```

```
            if (employees.containsKey(empName)) {
                allGroups.addAll(employees.get(empName).getGroups());
            }
        }

        if (allGroups.isEmpty()) {
            return null;
        }

        Group lca = findLCA(new ArrayList<>(allGroups));
        return lca != null ? lca.getName() : null;
    } finally {
        lock.readLock().unlock();
    }
}

// Usage example
public static void main(String[] args) {
    EmployeeDirectory directory = new EmployeeDirectory();
    directory.addGroup("Engineering", null);
    directory.addGroup("Backend", "Engineering");
    directory.addGroup("Frontend", "Engineering");
    directory.addEmployee("Alice", "Backend");
    directory.addEmployee("Bob", "Frontend");

    String result = directory.getCommonGroupForEmployees(
        Arrays.asList("Alice", "Bob")
    );
    System.out.println("Common group: " + result); // Output: Engineering
}
}
```

Problem 2: Tennis Court Booking System

Java Implementation with Priority Queues and Maintenance Logic

java

```java
import java.util.*;
import java.util.stream.Collectors;

class BookingRecord {
    private int id;
    private int startTime;
    private int finishTime;

    public BookingRecord(int id, int startTime, int finishTime) {
        this.id = id;
        this.startTime = startTime;
        this.finishTime = finishTime;
    }

    // Getters
    public int getId() { return id; }
    public int getStartTime() { return startTime; }
    public int getFinishTime() { return finishTime; }

    @Override
    public String toString() {
        return String.format("Booking{id=%d, start=%d, end=%d}", id, startTime, finishTime);
    }
}

class Court {
    private int id;
    private int availableTime;
    private int bookingsCount;
    private List<BookingRecord> bookings;

    public Court(int id) {
        this.id = id;
        this.availableTime = 0;
        this.bookingsCount = 0;
        this.bookings = new ArrayList<>();
    }

    // Getters and setters
    public int getId() { return id; }
    public int getAvailableTime() { return availableTime; }
    public void setAvailableTime(int availableTime) { this.availableTime = availableTime; }
    public int getBookingsCount() { return bookingsCount; }
    public void setBookingsCount(int bookingsCount) { this.bookingsCount = bookingsCount; }
    public List<BookingRecord> getBookings() { return bookings; }

    public void addBooking(BookingRecord booking) {
```

```
        bookings.add(booking);
        bookingsCount++;
    }

    @Override
    public String toString() {
        return String.format("Court{id=%d, available=%d, bookings=%d}",
                             id, availableTime, bookingsCount);
    }
}

public class TennisCourtScheduler {
    private List<Court> courts;
    private int courtCounter;

    public TennisCourtScheduler() {
        this.courts = new ArrayList<>();
        this.courtCounter = 0;
    }

    public List<Court> assignCourtsBasic(List<BookingRecord> bookingRecords) {
        // Sort bookings by start time
        List<BookingRecord> sortedBookings = bookingRecords.stream()
            .sorted(Comparator.comparingInt(BookingRecord::getStartTime))
            .collect(Collectors.toList());

        // Priority queue to track when courts become available
        PriorityQueue<Court> availableCourts = new PriorityQueue<>(
            Comparator.comparingInt(Court::getAvailableTime)
        );

        for (BookingRecord booking : sortedBookings) {
            Court court = null;

            // Check if any court is available
            if (!availableCourts.isEmpty() &&
                availableCourts.peek().getAvailableTime() <= booking.getStartTime()) {
                court = availableCourts.poll();
            } else {
                // Create new court
                court = new Court(courtCounter++);
                courts.add(court);
            }

            // Assign booking to court
            court.addBooking(booking);
            court.setAvailableTime(booking.getFinishTime());

            // Add court back to queue
            availableCourts.offer(court);
```

```
        availableCourts.offer(court);
    }

    return courts;
}

public List<Court> assignCourtsWithMaintenance(
        List<BookingRecord> bookingRecords,
        int maintenanceTime,
        int durability) {

    List<BookingRecord> sortedBookings = bookingRecords.stream()
        .sorted(Comparator.comparingInt(BookingRecord::getStartTime))
        .collect(Collectors.toList());

    PriorityQueue<Court> availableCourts = new PriorityQueue<>(
        Comparator.comparingInt(Court::getAvailableTime)
    );

    for (BookingRecord booking : sortedBookings) {
        Court suitableCourt = null;
        List<Court> tempCourts = new ArrayList<>();

        // Find available court that doesn't need maintenance
        while (!availableCourts.isEmpty()) {
            Court court = availableCourts.poll();

            if (court.getAvailableTime() <= booking.getStartTime() &&
                court.getBookingsCount() < durability) {
                suitableCourt = court;
                break;
            } else {
                tempCourts.add(court);
            }
        }

        // Restore remaining courts to queue
        availableCourts.addAll(tempCourts);

        if (suitableCourt == null) {
            // Create new court
            suitableCourt = new Court(courtCounter++);
            courts.add(suitableCourt);
        }

        // Assign booking
        suitableCourt.addBooking(booking);

        // Calculate next available time
        int nextAvailable = booking.getFinishTime();
```

```
            if (suitableCourt.getBookingsCount() >= durability) {
                nextAvailable += maintenanceTime;
                suitableCourt.setBookingsCount(0); // Reset after maintenance
            }

            suitableCourt.setAvailableTime(nextAvailable);
            availableCourts.offer(suitableCourt);
        }

        return courts;
    }

    public int minCourtsNeeded(List<BookingRecord> bookingRecords) {
        List<int[]> events = new ArrayList<>();

        // Create start and end events
        for (BookingRecord booking : bookingRecords) {
            events.add(new int[]{booking.getStartTime(), 1});  // Start event
            events.add(new int[]{booking.getFinishTime(), -1}); // End event
        }

        events.sort((a, b) -> {
            if (a[0] == b[0]) {
                return Integer.compare(a[1], b[1]); // End events before start events
            }
            return Integer.compare(a[0], b[0]);
        });

        int currentCourts = 0;
        int maxCourts = 0;

        for (int[] event : events) {
            currentCourts += event[1];
            maxCourts = Math.max(maxCourts, currentCourts);
        }

        return maxCourts;
    }

    public boolean checkBookingConflict(BookingRecord booking1, BookingRecord booking2) {
        return !(booking1.getFinishTime() <= booking2.getStartTime() ||
                 booking2.getFinishTime() <= booking1.getStartTime());
    }

    // Usage example
    public static void main(String[] args) {
        TennisCourtScheduler scheduler = new TennisCourtScheduler();
        List<BookingRecord> bookings = Arrays.asList(
            new BookingRecord(1, 9, 12),
```

```
            new BookingRecord(2, 10, 13),
            new BookingRecord(3, 11, 14),
            new BookingRecord(4, 15, 17)
        );

        List<Court> courts = scheduler.assignCourtsBasic(bookings);
        System.out.println("Number of courts needed: " + courts.size());

        courts.forEach(court -> {
            System.out.println(court + " -> " + court.getBookings());
        });

        System.out.println("Minimum courts needed: " + scheduler.minCourtsNeeded(bookings));
    }
}
```

Problem 3: Commodity Price Tracking

Java Implementation with Thread-Safe Operations

java

```java
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.locks.ReentrantLock;

public class CommodityPriceTracker {
    private final Map<Integer, Double> prices;
    private final PriorityQueue<PriceEntry> maxHeap;
    private final Set<PriceEntry> deletedEntries;
    private final ReentrantLock lock;
    private double currentMaxPrice;
    private Integer maxTimestamp;

    private static class PriceEntry {
        final double price;
        final int timestamp;

        PriceEntry(double price, int timestamp) {
            this.price = price;
            this.timestamp = timestamp;
        }

        @Override
        public boolean equals(Object obj) {
            if (this == obj) return true;
            if (obj == null || getClass() != obj.getClass()) return false;
            PriceEntry that = (PriceEntry) obj;
            return Double.compare(that.price, price) == 0 && timestamp == that.timestamp;
        }

        @Override
        public int hashCode() {
            return Objects.hash(price, timestamp);
        }

        @Override
        public String toString() {
            return String.format("PriceEntry{price=%.2f, timestamp=%d}", price, timestamp);
        }
    }

    public CommodityPriceTracker() {
        this.prices = new ConcurrentHashMap<>();
        this.maxHeap = new PriorityQueue<>((a, b) -> Double.compare(b.price, a.price));
        this.deletedEntries = ConcurrentHashMap.newKeySet();
        this.lock = new ReentrantLock();
        this.currentMaxPrice = 0.0;
        this.maxTimestamp = null;
```

```
}

public void updatePrice(int timestamp, double price) {
    lock.lock();
    try {
        Double oldPrice = prices.get(timestamp);
        prices.put(timestamp, price);

        // Add new entry to heap
        PriceEntry newEntry = new PriceEntry(price, timestamp);
        maxHeap.offer(newEntry);

        // If we had an old price, mark it as deleted
        if (oldPrice != null) {
            deletedEntries.add(new PriceEntry(oldPrice, timestamp));
        }

        // Update current max if necessary
        updateCurrentMax();
    } finally {
        lock.unlock();
    }
}

private void updateCurrentMax() {
    // Clean heap and update current max
    while (!maxHeap.isEmpty() && deletedEntries.contains(maxHeap.peek())) {
        PriceEntry removed = maxHeap.poll();
        deletedEntries.remove(removed);
    }

    if (!maxHeap.isEmpty()) {
        PriceEntry maxEntry = maxHeap.peek();
        currentMaxPrice = maxEntry.price;
        maxTimestamp = maxEntry.timestamp;
    } else {
        currentMaxPrice = 0.0;
        maxTimestamp = null;
    }
}

public double getMaxCommodityPrice() {
    lock.lock();
    try {
        updateCurrentMax();
        return currentMaxPrice;
    } finally {
        lock.unlock();
    }
}
```

```java
    }

    public Optional<Double> getPriceAtTimestamp(int timestamp) {
        return Optional.ofNullable(prices.get(timestamp));
    }

    public List<Map.Entry<Integer, Double>> getPriceHistorySorted() {
        return prices.entrySet().stream()
            .sorted(Map.Entry.comparingByKey())
            .collect(ArrayList::new, (list, entry) -> list.add(entry), ArrayList::addAll);
    }

    public Map<String, Object> getStatistics() {
        lock.lock();
        try {
            updateCurrentMax();
            OptionalDouble average = prices.values().stream()
                .mapToDouble(Double::doubleValue)
                .average();

            Map<String, Object> stats = new HashMap<>();
            stats.put("totalEntries", prices.size());
            stats.put("maxPrice", currentMaxPrice);
            stats.put("maxTimestamp", maxTimestamp);
            stats.put("averagePrice", average.orElse(0.0));

            return stats;
        } finally {
            lock.unlock();
        }
    }

    // Usage example
    public static void main(String[] args) {
        CommodityPriceTracker tracker = new CommodityPriceTracker();

        tracker.updatePrice(100, 25.5);
        tracker.updatePrice(105, 30.0);
        tracker.updatePrice(102, 28.0);
        tracker.updatePrice(100, 26.0); // Update existing timestamp

        System.out.println("Max price: " + tracker.getMaxCommodityPrice()); // 30.0
        System.out.println("Statistics: " + tracker.getStatistics());

        System.out.println("Price history:");
        tracker.getPriceHistorySorted().forEach(entry ->
            System.out.println("Timestamp: " + entry.getKey() + ", Price: " + entry.getValue())
        );
    }
}
```

Problem 4: Popular Content Tracking

Java Implementation with Efficient Max Tracking

java

```java
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.locks.ReentrantLock;

public class PopularContentTracker {
    private final Map<Integer, Integer> popularity;
    private final PriorityQueue<ContentEntry> maxHeap;
    private final Set<ContentEntry> deletedEntries;
    private final ReentrantLock lock;
    private int currentMaxPopularity;
    private int currentMaxContentId;

    private static class ContentEntry {
        final int popularity;
        final int contentId;

        ContentEntry(int popularity, int contentId) {
            this.popularity = popularity;
            this.contentId = contentId;
        }

        @Override
        public boolean equals(Object obj) {
            if (this == obj) return true;
            if (obj == null || getClass() != obj.getClass()) return false;
            ContentEntry that = (ContentEntry) obj;
            return popularity == that.popularity && contentId == that.contentId;
        }

        @Override
        public int hashCode() {
            return Objects.hash(popularity, contentId);
        }

        @Override
        public String toString() {
            return String.format("ContentEntry{popularity=%d, contentId=%d}",
                                 popularity, contentId);
        }
    }

    public PopularContentTracker() {
        this.popularity = new ConcurrentHashMap<>();
        this.maxHeap = new PriorityQueue<>((a, b) -> Integer.compare(b.popularity, a.popularity));
        this.deletedEntries = ConcurrentHashMap.newKeySet();
        this.lock = new ReentrantLock();
        this.currentMaxPopularity = 0;
```

```
    this.currentMaxContentId = -1;
}

public void increasePopularity(int contentId) {
    lock.lock();
    try {
        int oldPopularity = popularity.getOrDefault(contentId, 0);
        int newPopularity = oldPopularity + 1;
        popularity.put(contentId, newPopularity);

        // Mark old entry as deleted if it exists
        if (oldPopularity > 0) {
            deletedEntries.add(new ContentEntry(oldPopularity, contentId));
        }

        // Add new entry to heap
        ContentEntry newEntry = new ContentEntry(newPopularity, contentId);
        maxHeap.offer(newEntry);

        // Update current max if this is now the highest
        if (newPopularity > currentMaxPopularity) {
            currentMaxPopularity = newPopularity;
            currentMaxContentId = contentId;
        }
    } finally {
        lock.unlock();
    }
}

public void decreasePopularity(int contentId) {
    lock.lock();
    try {
        if (!popularity.containsKey(contentId) || popularity.get(contentId) <= 0) {
            return;
        }

        int oldPopularity = popularity.get(contentId);
        int newPopularity = oldPopularity - 1;

        // Mark old entry as deleted
        deletedEntries.add(new ContentEntry(oldPopularity, contentId));

        // Update popularity map
        if (newPopularity > 0) {
            popularity.put(contentId, newPopularity);
            maxHeap.offer(new ContentEntry(newPopularity, contentId));
        } else {
            popularity.put(contentId, 0);
        }
```

```
        // If this was the max content, we need to recalculate
        if (contentId == currentMaxContentId && oldPopularity == currentMaxPopularity) {
            recalculateMax();
        }
    } finally {
        lock.unlock();
    }
}

private void recalculateMax() {
    // Clean up deleted entries from heap
    while (!maxHeap.isEmpty() && deletedEntries.contains(maxHeap.peek())) {
        ContentEntry removed = maxHeap.poll();
        deletedEntries.remove(removed);
    }

    if (!maxHeap.isEmpty()) {
        ContentEntry maxEntry = maxHeap.peek();
        currentMaxPopularity = maxEntry.popularity;
        currentMaxContentId = maxEntry.contentId;
    } else {
        // Find max from current popularity map
        int maxPop = 0;
        int maxContent = -1;

        for (Map.Entry<Integer, Integer> entry : popularity.entrySet()) {
            if (entry.getValue() > maxPop) {
                maxPop = entry.getValue();
                maxContent = entry.getKey();
            }
        }

        currentMaxPopularity = maxPop;
        currentMaxContentId = maxPop > 0 ? maxContent : -1;
    }
}

public int getMostPopularContent() {
    lock.lock();
    try {
        recalculateMax();
        return currentMaxPopularity > 0 ? currentMaxContentId : -1;
    } finally {
        lock.unlock();
    }
}

public int getPopularity(int contentId) {
    return popularity.getOrDefault(contentId, 0);
```

```java
    }

    public List<Map.Entry<Integer, Integer>> getTopContent(int limit) {
        lock.lock();
        try {
            return popularity.entrySet().stream()
                .filter(entry -> entry.getValue() > 0)
                .sorted((a, b) -> Integer.compare(b.getValue(), a.getValue()))
                .limit(limit)
                .collect(ArrayList::new, (list, entry) -> list.add(entry), ArrayList::addAll);
        } finally {
            lock.unlock();
        }
    }

    // Usage example
    public static void main(String[] args) {
        PopularContentTracker tracker = new PopularContentTracker();

        tracker.increasePopularity(1);
        tracker.increasePopularity(2);
        tracker.increasePopularity(1);
        tracker.decreasePopularity(2);

        System.out.println("Most popular content: " + tracker.getMostPopularContent()); // Should return 1
        System.out.println("Content 1 popularity: " + tracker.getPopularity(1));
        System.out.println("Content 2 popularity: " + tracker.getPopularity(2));

        System.out.println("Top content:");
        tracker.getTopContent(5).forEach(entry ->
            System.out.println("Content " + entry.getKey() + ": " + entry.getValue())
        );
    }
}
```

Problem 5: Weighted Graph - Shortest Path

Java Implementation with Dijkstra's Algorithm

java

```java
import java.util.*;

public class WeightedGraph {
    private final Map<String, Map<String, Integer>> graph;
    private final Set<String> nodes;

    public WeightedGraph() {
        this.graph = new HashMap<>();
        this.nodes = new HashSet<>();
    }

    public void addEdge(String source, String destination, int weight) {
        graph.computeIfAbsent(source, k -> new HashMap<>()).put(destination, weight);
        nodes.add(source);
        nodes.add(destination);
    }

    public boolean canReach(String source, String destination) {
        if (!nodes.contains(source) || !nodes.contains(destination)) {
            return false;
        }

        if (source.equals(destination)) {
            return true;
        }

        // BFS to check reachability
        Set<String> visited = new HashSet<>();
        Queue<String> queue = new LinkedList<>();
        queue.offer(source);

        while (!queue.isEmpty()) {
            String current = queue.poll();

            if (current.equals(destination)) {
                return true;
            }

            if (visited.contains(current)) {
                continue;
            }

            visited.add(current);

            if (graph.containsKey(current)) {
                for (String neighbor : graph.get(current).keySet()) {
                    if (!visited.contains(neighbor)) {
```

```
                queue.offer(neighbor);
            }
        }
    }

    return false;
}

public Optional<Integer> shortestPathTime(String source, String destination) {
    if (!nodes.contains(source) || !nodes.contains(destination)) {
        return Optional.empty();
    }

    if (source.equals(destination)) {
        return Optional.of(0);
    }

    // Dijkstra's algorithm
    Map<String, Integer> distances = new HashMap<>();
    PriorityQueue<NodeDistance> pq = new PriorityQueue<>(
        Comparator.comparingInt(nd -> nd.distance)
    );
    Set<String> visited = new HashSet<>();

    // Initialize distances
    for (String node : nodes) {
        distances.put(node, Integer.MAX_VALUE);
    }
    distances.put(source, 0);
    pq.offer(new NodeDistance(source, 0));

    while (!pq.isEmpty()) {
        NodeDistance current = pq.poll();
        String currentNode = current.node;
        int currentDist = current.distance;

        if (visited.contains(currentNode)) {
            continue;
        }

        visited.add(currentNode);

        if (currentNode.equals(destination)) {
            return Optional.of(currentDist);
        }

        // Check all neighbors
        if (graph.containsKey(currentNode)) {
            for (Map.Entry<String, Integer> edge : graph.get(currentNode).entrySet()) {
```

```
            String neighbor = edge.getKey();
            int weight = edge.getValue();

            if (!visited.contains(neighbor)) {
                int newDistance = currentDist + weight;
                if (newDistance < distances.get(neighbor)) {
                    distances.put(neighbor, newDistance);
                    pq.offer(new NodeDistance(neighbor, newDistance));
                }
            }
        }
    }

    // Reconstruct path
    if (distances.get(destination) == Integer.MAX_VALUE) {
        return new ShortestPathResult(Optional.empty(), Collections.emptyList());
    }

    List<String> path = new ArrayList<>();
    String current = destination;
    while (current != null) {
        path.add(current);
        current = previous.get(current);
    }
    Collections.reverse(path);

    return new ShortestPathResult(Optional.of(distances.get(destination)), path);
}

// Helper classes
private static class NodeDistance {
    final String node;
    final int distance;

    NodeDistance(String node, int distance) {
        this.node = node;
        this.distance = distance;
    }
}

public static class ShortestPathResult {
    private final Optional<Integer> distance;
    private final List<String> path;

    public ShortestPathResult(Optional<Integer> distance, List<String> path) {
        this.distance = distance;
        this.path = new ArrayList<>(path);
    }
```

```
    public Optional<Integer> getDistance() { return distance; }
    public List<String> getPath() { return new ArrayList<>(path); }

    @Override
    public String toString() {
        return String.format("ShortestPathResult{distance=%s, path=%s}",
                distance.map(String::valueOf).orElse("No path"), path);
    }
}

// Advanced features
public List<String> getAllNodes() {
    return new ArrayList<>(nodes);
}

public Map<String, Integer> getAllPaths(String source) {
    Map<String, Integer> allDistances = new HashMap<>();

    if (!nodes.contains(source)) {
        return allDistances;
    }

    // Run Dijkstra from source to all nodes
    Map<String, Integer> distances = new HashMap<>();
    PriorityQueue<NodeDistance> pq = new PriorityQueue<>(
        Comparator.comparingInt(nd -> nd.distance)
    );
    Set<String> visited = new HashSet<>();

    for (String node : nodes) {
        distances.put(node, Integer.MAX_VALUE);
    }
    distances.put(source, 0);
    pq.offer(new NodeDistance(source, 0));

    while (!pq.isEmpty()) {
        NodeDistance current = pq.poll();
        String currentNode = current.node;

        if (visited.contains(currentNode)) {
            continue;
        }

        visited.add(currentNode);
        allDistances.put(currentNode, distances.get(currentNode));

        if (graph.containsKey(currentNode)) {
            for (Map.Entry<String, Integer> edge : graph.get(currentNode).entrySet()) {
```

```
                String neighbor = edge.getKey();
                int weight = edge.getValue();

                if (!visited.contains(neighbor)) {
                    int newDistance = distances.get(currentNode) + weight;
                    if (newDistance < distances.get(neighbor)) {
                        distances.put(neighbor, newDistance);
                        pq.offer(new NodeDistance(neighbor, newDistance));
                    }
                }
            }
        }
    }

    return allDistances;
}

// Usage example
public static void main(String[] args) {
    WeightedGraph graph = new WeightedGraph();
    graph.addEdge("A", "B", 100);
    graph.addEdge("B", "C", 200);
    graph.addEdge("A", "C", 350);
    graph.addEdge("C", "D", 150);

    System.out.println("Can reach A to D: " + graph.canReach("A", "D"));
    System.out.println("Shortest time A to D: " + graph.shortestPathTime("A", "D"));

    ShortestPathResult result = graph.getShortestPath("A", "D");
    System.out.println("Shortest path result: " + result);

    System.out.println("All distances from A:");
    graph.getAllPaths("A").forEach((node, distance) ->
        System.out.println(node + ": " + (distance == Integer.MAX_VALUE ? "unreachable" : distance + "ms"))
    );
}
}
```

Problem 6: Job Interval Reporting

Java Implementation for CI Pipeline Analysis

java

```java
import java.util.*;
import java.util.stream.Collectors;

public class JobIntervalReporter {

    public static class Interval {
        private final int start;
        private final int end;

        public Interval(int start, int end) {
            this.start = start;
            this.end = end;
        }

        public int getStart() { return start; }
        public int getEnd() { return end; }

        public boolean overlaps(Interval other) {
            return this.start < other.end && other.start < this.end;
        }

        public Interval merge(Interval other) {
            return new Interval(Math.min(this.start, other.start),
                                Math.max(this.end, other.end));
        }

        public int duration() {
            return end - start;
        }

        @Override
        public String toString() {
            return String.format("[%d, %d]", start, end);
        }

        @Override
        public boolean equals(Object obj) {
            if (this == obj) return true;
            if (obj == null || getClass() != obj.getClass()) return false;
            Interval interval = (Interval) obj;
            return start == interval.start && end == interval.end;
        }

        @Override
        public int hashCode() {
            return Objects.hash(start, end);
        }
```

```
}

public List<Interval> mergeIntervals(List<Interval> intervals) {
    if (intervals == null || intervals.isEmpty()) {
        return new ArrayList<>();
    }

    // Sort intervals by start time
    List<Interval> sortedIntervals = intervals.stream()
        .sorted(Comparator.comparingInt(Interval::getStart))
        .collect(Collectors.toList());

    List<Interval> merged = new ArrayList<>();
    Interval current = sortedIntervals.get(0);

    for (int i = 1; i < sortedIntervals.size(); i++) {
        Interval next = sortedIntervals.get(i);

        if (current.getEnd() >= next.getStart()) {
            // Overlapping intervals - merge them
            current = current.merge(next);
        } else {
            // Non-overlapping - add current and move to next
            merged.add(current);
            current = next;
        }
    }

    merged.add(current); // Don't forget the last interval
    return merged;
}

public List<Interval> findActiveWindows(List<Interval> pipelineWindows) {
    return mergeIntervals(pipelineWindows);
}

public int getTotalActiveTime(List<Interval> pipelineWindows) {
    List<Interval> merged = mergeIntervals(pipelineWindows);
    return merged.stream()
        .mapToInt(Interval::duration)
        .sum();
}

public List<Interval> findIdlePeriods(List<Interval> pipelineWindows, Interval totalRange) {
    List<Interval> merged = mergeIntervals(pipelineWindows);
    List<Interval> idlePeriods = new ArrayList<>();

    if (merged.isEmpty()) {
        idlePeriods.add(totalRange);
        return idlePeriods;
```

```
    }

    // Check before first interval
    if (merged.get(0).getStart() > totalRange.getStart()) {
        idlePeriods.add(new Interval(totalRange.getStart(), merged.get(0).getStart()));
    }

    // Check between intervals
    for (int i = 0; i < merged.size() - 1; i++) {
        int gapStart = merged.get(i).getEnd();
        int gapEnd = merged.get(i + 1).getStart();
        if (gapStart < gapEnd) {
            idlePeriods.add(new Interval(gapStart, gapEnd));
        }
    }

    // Check after last interval
    Interval lastInterval = merged.get(merged.size() - 1);
    if (lastInterval.getEnd() < totalRange.getEnd()) {
        idlePeriods.add(new Interval(lastInterval.getEnd(), totalRange.getEnd()));
    }

    return idlePeriods;
}

public PeakConcurrencyResult findPeakConcurrencyTime(List<Interval> pipelineWindows) {
    if (pipelineWindows == null || pipelineWindows.isEmpty()) {
        return new PeakConcurrencyResult(0, Collections.emptyList());
    }

    List<TimeEvent> events = new ArrayList<>();

    // Create start and end events
    for (Interval interval : pipelineWindows) {
        events.add(new TimeEvent(interval.getStart(), 1));   // Start event
        events.add(new TimeEvent(interval.getEnd(), -1));    // End event
    }

    // Sort events by time, with end events before start events at same time
    events.sort((a, b) -> {
        if (a.time == b.time) {
            return Integer.compare(a.type, b.type); // -1 (end) before 1 (start)
        }
        return Integer.compare(a.time, b.time);
    });

    int currentCount = 0;
    int maxCount = 0;
    List<Interval> maxPeriods = new ArrayList<>();
```

```
    Integer periodStart = null;

    for (int i = 0; i < events.size(); i++) {
        int time = events.get(i).time;

        // Process all events at the same time
        while (i < events.size() && events.get(i).time == time) {
            currentCount += events.get(i).type;
            i++;
        }
        i--; // Adjust for the outer loop increment

        // Check if we have a new maximum
        if (currentCount > maxCount) {
            maxCount = currentCount;
            maxPeriods.clear();
            periodStart = time;
        }

        // If count drops below max, end the current period
        if (currentCount < maxCount && periodStart != null) {
            maxPeriods.add(new Interval(periodStart, time));
            periodStart = null;
        } else if (currentCount == maxCount && periodStart == null) {
            periodStart = time;
        }
    }

    return new PeakConcurrencyResult(maxCount, maxPeriods);
}

public PipelineEfficiencyReport analyzeEfficiency(List<Interval> pipelineWindows) {
    if (pipelineWindows == null || pipelineWindows.isEmpty()) {
        return new PipelineEfficiencyReport(0, 0, 0, 0.0, 0, 0.0);
    }

    int totalRuntime = pipelineWindows.stream()
        .mapToInt(Interval::duration)
        .sum();

    int activeTime = getTotalActiveTime(pipelineWindows);
    PeakConcurrencyResult peakResult = findPeakConcurrencyTime(pipelineWindows);
    double avgDuration = (double) totalRuntime / pipelineWindows.size();
    double efficiency = activeTime > 0 ? (double) activeTime / totalRuntime : 0.0;

    return new PipelineEfficiencyReport(
        pipelineWindows.size(),
        totalRuntime,
        activeTime,
```

```
            efficiency,
            peakResult.getMaxConcurrency(),
            avgDuration
        );
    }

    // Helper classes
    private static class TimeEvent {
        final int time;
        final int type; // 1 for start, -1 for end

        TimeEvent(int time, int type) {
            this.time = time;
            this.type = type;
        }
    }

    public static class PeakConcurrencyResult {
        private final int maxConcurrency;
        private final List<Interval> peakPeriods;

        public PeakConcurrencyResult(int maxConcurrency, List<Interval> peakPeriods) {
            this.maxConcurrency = maxConcurrency;
            this.peakPeriods = new ArrayList<>(peakPeriods);
        }

        public int getMaxConcurrency() { return maxConcurrency; }
        public List<Interval> getPeakPeriods() { return new ArrayList<>(peakPeriods); }

        @Override
        public String toString() {
            return String.format("PeakConcurrencyResult{maxConcurrency=%d, peakPeriods=%s}",
                                 maxConcurrency, peakPeriods);
        }
    }

    public static class PipelineEfficiencyReport {
        private final int totalPipelines;
        private final int totalRuntime;
        private final int activeTime;
        private final double efficiency;
        private final int maxConcurrency;
        private final double avgPipelineDuration;

        public PipelineEfficiencyReport(int totalPipelines, int totalRuntime, int activeTime,
                                        double efficiency, int maxConcurrency, double avgPipelineDuration) {
            this.totalPipelines = totalPipelines;
            this.totalRuntime = totalRuntime;
            this.activeTime = activeTime;
            this.efficiency = efficiency;
```

```
        this.maxConcurrency = maxConcurrency;
        this.avgPipelineDuration = avgPipelineDuration;
    }

    // Getters
    public int getTotalPipelines() { return totalPipelines; }
    public int getTotalRuntime() { return totalRuntime; }
    public int getActiveTime() { return activeTime; }
    public double getEfficiency() { return efficiency; }
    public int getMaxConcurrency() { return maxConcurrency; }
    public double getAvgPipelineDuration() { return avgPipelineDuration; }

    @Override
    public String toString() {
        return String.format(
            "PipelineEfficiencyReport{\n" +
            "  totalPipelines=%d,\n" +
            "  totalRuntime=%d,\n" +
            "  activeTime=%d,\n" +
            "  efficiency=%.2f%%,\n" +
            "  maxConcurrency=%d,\n" +
            "  avgPipelineDuration=%.2f\n" +
            "}",
            totalPipelines, totalRuntime, activeTime,
            efficiency * 100, maxConcurrency, avgPipelineDuration
        );
    }
}

// Usage example
public static void main(String[] args) {
    JobIntervalReporter reporter = new JobIntervalReporter();

    // Example from the problem: [{2, 5}, {12, 15}, {4, 8}]
    List<Interval> pipelineWindows = Arrays.asList(
        new Interval(2, 5),
        new Interval(12, 15),
        new Interval(4, 8)
    );

    List<Interval> activeWindows = reporter.findActiveWindows(pipelineWindows);
    System.out.println("Active windows: " + activeWindows); // Output: [[2, 8], [12, 15]]

    int totalActive = reporter.getTotalActiveTime(pipelineWindows);
    System.out.println("Total active time: " + totalActive);

    PeakConcurrencyResult peakResult = reporter.findPeakConcurrencyTime(pipelineWindows);
    System.out.println("Peak concurrency: " + peakResult);
```

```
        List<Interval> idlePeriods = reporter.findIdlePeriods(pipelineWindows, new Interval(0, 20));
        System.out.println("Idle periods: " + idlePeriods);

        PipelineEfficiencyReport efficiency = reporter.analyzeEfficiency(pipelineWindows);
        System.out.println("Efficiency report:\n" + efficiency);
    }
}
```

Java-Specific Interview Preparation Topics

1. **Core Java Concepts for Senior/Lead Roles**

Essential Java Features:

java

```java
// Streams and Functional Programming
public class StreamExamples {
    public List<String> processEmployees(List<Employee> employees) {
        return employees.stream()
            .filter(emp -> emp.getDepartment().equals("Engineering"))
            .filter(emp -> emp.getYearsExperience() > 5)
            .sorted(Comparator.comparing(Employee::getName))
            .map(Employee::getEmail)
            .collect(Collectors.toList());
    }

    // Parallel processing for large datasets
    public Map<String, Long> getDepartmentCounts(List<Employee> employees) {
        return employees.parallelStream()
            .collect(Collectors.groupingBy(
                Employee::getDepartment,
                Collectors.counting()
            ));
    }
}

// Advanced Collections Usage
public class CollectionsExamples {
    // Thread-safe collections
    private final ConcurrentHashMap<String, AtomicInteger> counters = new ConcurrentHashMap<>();
    private final CopyOnWriteArrayList<String> auditLog = new CopyOnWriteArrayList<>();

    // Custom comparators and sorting
    public void sortEmployeesByMultipleCriteria(List<Employee> employees) {
        employees.sort(
            Comparator.comparing(Employee::getDepartment)
                .thenComparing(Employee::getLevel)
                .thenComparing(Employee::getName)
        );
    }
}

// Memory Management and Performance
public class MemoryOptimization {
    // Use primitive collections to avoid boxing overhead
    private final TIntObjectHashMap<String> idToName = new TIntObjectHashMap<>();

    // Object pooling for frequent allocations
    private final ObjectPool<StringBuilder> stringBuilderPool =
        new GenericObjectPool<>(new StringBuilderFactory());

    // Weak references for caches
```

```
    private final WeakHashMap<String, ExpensiveObject> cache = new WeakHashMap<>();
}
```

2. Concurrency and Multithreading

java

```java
// Advanced concurrency patterns
public class ConcurrencyPatterns {

    // Producer-Consumer with BlockingQueue
    public class TaskProcessor {
        private final BlockingQueue<Task> taskQueue = new ArrayBlockingQueue<>(1000);
        private final ExecutorService executorService = Executors.newFixedThreadPool(10);

        public void startProcessing() {
            for (int i = 0; i < 5; i++) {
                executorService.submit(this::processTasks);
            }
        }

        private void processTasks() {
            while (!Thread.currentThread().isInterrupted()) {
                try {
                    Task task = taskQueue.take();
                    processTask(task);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    break;
                }
            }
        }
    }

    // Lock-free data structures
    public class LockFreeCounter {
        private final AtomicLong counter = new AtomicLong(0);

        public long increment() {
            return counter.incrementAndGet();
        }

        public long get() {
            return counter.get();
        }
    }

    // CompletableFuture for async processing
    public class AsyncProcessor {
        public CompletableFuture<String> processDataAsync(String input) {
            return CompletableFuture
                .supplyAsync(() -> heavyComputation(input))
                .thenApply(this::transform)
                .thenCompose(this::validate)
```

```
            .exceptionally(this::handleError);
    }

    public CompletableFuture<List<String>> processMultipleAsync(List<String> inputs) {
        List<CompletableFuture<String>> futures = inputs.stream()
            .map(this::processDataAsync)
            .collect(Collectors.toList());

        return CompletableFuture.allOf(futures.toArray(new CompletableFuture[0]))
            .thenApply(v -> futures.stream()
                .map(CompletableFuture::join)
                .collect(Collectors.toList()));
    }
  }
}
```

3. Spring Framework & Enterprise Java

java

```java
// Spring Boot application structure
@SpringBootApplication
@EnableAsync
@EnableScheduling
public class AtlassianStyleApplication {

    @Bean
    @Primary
    public TaskExecutor taskExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(10);
        executor.setMaxPoolSize(50);
        executor.setQueueCapacity(100);
        executor.setThreadNamePrefix("async-task-");
        executor.initialize();
        return executor;
    }
}

// Service layer with proper error handling
@Service
@Transactional
public class EmployeeService {

    private final EmployeeRepository employeeRepository;
    private final NotificationService notificationService;
    private final CacheManager cacheManager;

    public EmployeeService(EmployeeRepository employeeRepository,
                           NotificationService notificationService,
                           CacheManager cacheManager) {
        this.employeeRepository = employeeRepository;
        this.notificationService = notificationService;
        this.cacheManager = cacheManager;
    }

    @Cacheable(value = "employees", key = "#id")
    public Optional<Employee> findById(Long id) {
        return employeeRepository.findById(id);
    }

    @Async
    @Retryable(value = {Exception.class}, maxAttempts = 3, backoff = @Backoff(delay = 1000))
    public CompletableFuture<Void> processEmployeeAsync(Long employeeId) {
        try {
            Employee employee = findById(employeeId)
                .orElseThrow(() -> new EmployeeNotFoundException("Employee not found: " + employeeId));
```

```
            // Heavy processing
            processEmployee(employee);

            // Send notification
            notificationService.sendNotification(employee);

            return CompletableFuture.completedFuture(null);
        } catch (Exception e) {
            log.error("Error processing employee: {}", employeeId, e);
            throw e;
        }
    }

    @EventListener
    public void handleEmployeeCreated(EmployeeCreatedEvent event) {
        // Clear related caches
        cacheManager.getCache("departments").evict(event.getEmployee().getDepartmentId());
    }
}

// REST API with proper validation and error handling
@RestController
@RequestMapping("/api/v1/employees")
@Validated
public class EmployeeController {

    private final EmployeeService employeeService;

    @GetMapping("/{id}")
    public ResponseEntity<EmployeeDto> getEmployee(
            @PathVariable @Min(1) Long id,
            HttpServletRequest request) {

        Optional<Employee> employee = employeeService.findById(id);

        if (employee.isEmpty()) {
            return ResponseEntity.notFound().build();
        }

        EmployeeDto dto = EmployeeMapper.toDto(employee.get());

        // Add HATEOAS links
        dto.add(linkTo(methodOn(EmployeeController.class).getEmployee(id)).withSelfRel());
        dto.add(linkTo(methodOn(DepartmentController.class)
            .getDepartment(employee.get().getDepartmentId())).withRel("department"));

        return ResponseEntity.ok()
            .cacheControl(CacheControl.maxAge(Duration.ofMinutes(5)))
            .body(dto);
```

```
    }

    @PostMapping
    public ResponseEntity<EmployeeDto> createEmployee(
            @RequestBody @Valid CreateEmployeeRequest request,
            UriComponentsBuilder uriBuilder) {

        Employee employee = employeeService.createEmployee(request);
        EmployeeDto dto = EmployeeMapper.toDto(employee);

        URI location = uriBuilder.path("/api/v1/employees/{id}")
            .buildAndExpand(employee.getId())
            .toUri();

        return ResponseEntity.created(location).body(dto);
    }
}
```

4. Database & JPA Best Practices

java

```java
// Optimized JPA entities
@Entity
@Table(name = "employees", indexes = {
    @Index(name = "idx_employee_email", columnList = "email"),
    @Index(name = "idx_employee_dept_status", columnList = "department_id, status")
})
@EntityListeners(AuditingEntityListener.class)
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String email;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "department_id")
    private Department department;

    @OneToMany(mappedBy = "employee", cascade = CascadeType.ALL, orphanRemoval = true)
    @BatchSize(size = 20)
    private List<Project> projects = new ArrayList<>();

    @CreatedDate
    private LocalDateTime createdAt;

    @LastModifiedDate
    private LocalDateTime updatedAt;

    @Version
    private Long version;
}

// Repository with custom queries
@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long>, JpaSpecificationExecutor<Employee> {

    @Query("SELECT e FROM Employee e JOIN FETCH e.department WHERE e.status = :status")
    List<Employee> findActiveEmployeesWithDepartment(@Param("status") EmployeeStatus status);

    @Query(value = "SELECT * FROM employees e WHERE e.hire_date >= :fromDate",
           countQuery = "SELECT count(*) FROM employees e WHERE e.hire_date >= :fromDate",
           nativeQuery = true)
    Page<Employee> findRecentHires(@Param("fromDate") LocalDate fromDate, Pageable pageable);

    @Modifying
```

```
    @Query("UPDATE Employee e SET e.status = :newStatus WHERE e.department.id = :deptId")
    int bulkUpdateEmployeeStatus(@Param("deptId") Long departmentId,
                                 @Param("newStatus") EmployeeStatus newStatus);
}

// Database migration management
@Component
public class DatabaseMigrations {

    @EventListener(ApplicationReadyEvent.class)
    public void performDataMigration() {
        // Custom data migration logic
        migrateOldEmployeeData();
    }

    @Transactional
    public void migrateOldEmployeeData() {
        // Batch processing for large datasets
        int batchSize = 1000;
        int offset = 0;

        List<OldEmployee> batch;
        do {
            batch = oldEmployeeRepository.findBatch(offset, batchSize);

            List<Employee> newEmployees = batch.stream()
                .map(this::convertToNewFormat)
                .collect(Collectors.toList());

            employeeRepository.saveAll(newEmployees);

            offset += batchSize;
        } while (!batch.isEmpty());
    }
}
```

5. Testing Strategies for Java

java

```java
// Unit testing with Mockito and JUnit 5
@ExtendWith(MockitoExtension.class)
class EmployeeServiceTest {

    @Mock
    private EmployeeRepository employeeRepository;

    @Mock
    private NotificationService notificationService;

    @InjectMocks
    private EmployeeService employeeService;

    @Test
    void shouldCreateEmployeeSuccessfully() {
        // Given
        CreateEmployeeRequest request = new CreateEmployeeRequest("john@example.com", "Engineering");
        Employee savedEmployee = new Employee();
        savedEmployee.setId(1L);
        savedEmployee.setEmail("john@example.com");

        when(employeeRepository.save(any(Employee.class))).thenReturn(savedEmployee);

        // When
        Employee result = employeeService.createEmployee(request);

        // Then
        assertThat(result.getId()).isEqualTo(1L);
        assertThat(result.getEmail()).isEqualTo("john@example.com");

        verify(employeeRepository).save(argThat(emp ->
            emp.getEmail().equals("john@example.com")));
        verify(notificationService).sendWelcomeEmail(savedEmployee);
    }

    @Test
    void shouldThrowExceptionWhenEmployeeNotFound() {
        // Given
        when(employeeRepository.findById(999L)).thenReturn(Optional.empty());

        // When & Then
        assertThatThrownBy(() -> employeeService.getEmployee(999L))
            .isInstanceOf(EmployeeNotFoundException.class)
            .hasMessageContaining("Employee not found: 999");
    }
}
```

```java
// Integration testing with TestContainers
@SpringBootTest
@Testcontainers
class EmployeeIntegrationTest {

    @Container
    static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>("postgres:13")
            .withDatabaseName("testdb")
            .withUsername("test")
            .withPassword("test");

    @DynamicPropertySource
    static void configureProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.datasource.url", postgres::getJdbcUrl);
        registry.add("spring.datasource.username", postgres::getUsername);
        registry.add("spring.datasource.password", postgres::getPassword);
    }

    @Autowired
    private TestRestTemplate restTemplate;

    @Autowired
    private EmployeeRepository employeeRepository;

    @Test
    void shouldCreateAndRetrieveEmployee() {
        // Given
        CreateEmployeeRequest request = new CreateEmployeeRequest("jane@example.com", "Engineering");

        // When - Create employee
        ResponseEntity<EmployeeDto> createResponse = restTemplate.postForEntity(
            "/api/v1/employees", request, EmployeeDto.class);

        // Then - Verify creation
        assertThat(createResponse.getStatusCode()).isEqualTo(HttpStatus.CREATED);
        EmployeeDto createdEmployee = createResponse.getBody();
        assertThat(createdEmployee.getEmail()).isEqualTo("jane@example.com");

        // When - Retrieve employee
        ResponseEntity<EmployeeDto> getResponse = restTemplate.getForEntity(
            "/api/v1/employees/" + createdEmployee.getId(), EmployeeDto.class);

        // Then - Verify retrieval
        assertThat(getResponse.getStatusCode()).isEqualTo(new NodeDistance(neighbor, newDistance));
                }
            }
        }
    }
}
```

```
        return Optional.empty(); // No path found
    }

    public ShortestPathResult getShortestPath(String source, String destination) {
        if (!nodes.contains(source) || !nodes.contains(destination)) {
            return new ShortestPathResult(Optional.empty(), Collections.emptyList());
        }

        if (source.equals(destination)) {
            return new ShortestPathResult(Optional.of(0), Arrays.asList(source));
        }

        Map<String, Integer> distances = new HashMap<>();
        Map<String, String> previous = new HashMap<>();
        PriorityQueue<NodeDistance> pq = new PriorityQueue<>(
            Comparator.comparingInt(nd -> nd.distance)
        );
        Set<String> visited = new HashSet<>();

        // Initialize
        for (String node : nodes) {
            distances.put(node, Integer.MAX_VALUE);
            previous.put(node, null);
        }
        distances.put(source, 0);
        pq.offer(new NodeDistance(source, 0));

        while (!pq.isEmpty()) {
            NodeDistance current = pq.poll();
            String currentNode = current.node;

            if (visited.contains(currentNode)) {
                continue;
            }

            visited.add(currentNode);

            if (graph.containsKey(currentNode)) {
                for (Map.Entry<String, Integer> edge : graph.get(currentNode).entrySet()) {
                    String neighbor = edge.getKey();
                    int weight = edge.getValue();

                    if (!visited.contains(neighbor)) {
                        int newDistance = distances.get(currentNode) + weight;
                        if (newDistance < distances.get(neighbor)) {
                            distances.put(neighbor, newDistance);
                            previous.put(neighbor, currentNode);
                            pq.offer(
```