IQRA UNIVERSITY

DSA

OPEN ENDED LAB

GROUP MEMBERS:

KAMRAN HASSAN (63976)

MARYUM AMIR (63932)

SANA ARSHAD (63910)

DEPARTMENT:

BS SOFTWARE ENGINEERING

INSTRUCTOR:

MUHAMMAD KAMRAN

Application: Dynamic Event Scheduler

1. Proposal Submission (Problem Definition)

Problem Description: A dynamic event scheduling system where users can create, view, and manage events. Each event will have attributes like title, start time, end time, and priority. The system will handle overlapping events, prioritize events, and optimize scheduling.

Chosen Data Structure:

LinkedList:

- Allows efficient insertion and deletion operations.
- Ideal for managing dynamic collections of events where sequential access is required.

Algorithmic Challenges:

- Efficiently sorting events by priority or start time.
- Detecting and managing overlapping events.
- Optimizing insertion/removal operations in a dynamic list.

2. Data Structure Design

Custom Data Structure: Event

We use a class Event to represent individual events. Each event has:

Attributes: Title, start time, end time, and priority.

▪ Code for Event Class:

```
import java.time.LocalTime;
class Event {
    String title;
    LocalTime startTime;
    LocalTime endTime;
    int priority;

    public Event(String title, LocalTime startTime, LocalTime
endTime, int priority) {
        this.title = title;
        this.startTime = startTime;
```

```
        this.endTime = endTime;
        this.priority = priority;
    }

    @Override
    public String toString() {
        return "Event{" +
               "title='" + title + '\'' +
               ", startTime=" + startTime +
               ", endTime=" + endTime +
               ", priority=" + priority +
               '}';
    }
}
```

Primary Data Structure: LinkedList

- Store all events in a LinkedList for dynamic insertion and deletion.
- Use methods like add() and remove() for managing events dynamically.

3. Algorithm Development

Algorithms to Handle Operations

- **Add an Event:**

```
public void addEvent(Event event) {
    events.add(event);
    events.sort((e1, e2) -> e1.priority - e2.priority); // Sort by priority
}
```

Detect Overlapping Events:

```
public List<Event> getOverlappingEvents() {
    List<Event> overlapping = new LinkedList<>();
```

```
    for (int i = 0; i < events.size(); i++) {
        for (int j = i + 1; j < events.size(); j++) {
            if
(events.get(i).endTime.isAfter(events.get(j).startTime)) {
                overlapping.add(events.get(i));
                overlapping.add(events.get(j));
            }
        }
    }
    return overlapping;
}
```

- **Search Events by Title:**

```
public Event searchEvent(String title) {
    for (Event event : events) {
        if (event.title.equalsIgnoreCase(title)) {
            return event;
        }
    }
    return null;
}
```

- **Remove an Event:**

```
public void removeEvent(String title) {
    events.removeIf(event ->
event.title.equalsIgnoreCase(title));
}
```

4. Integration and Optimization

Integration of Features

Combine all functionalities into a cohesive application:

- Add new events.
- View all events.
- Search events by title.
- Detect overlapping events.
- Remove events.

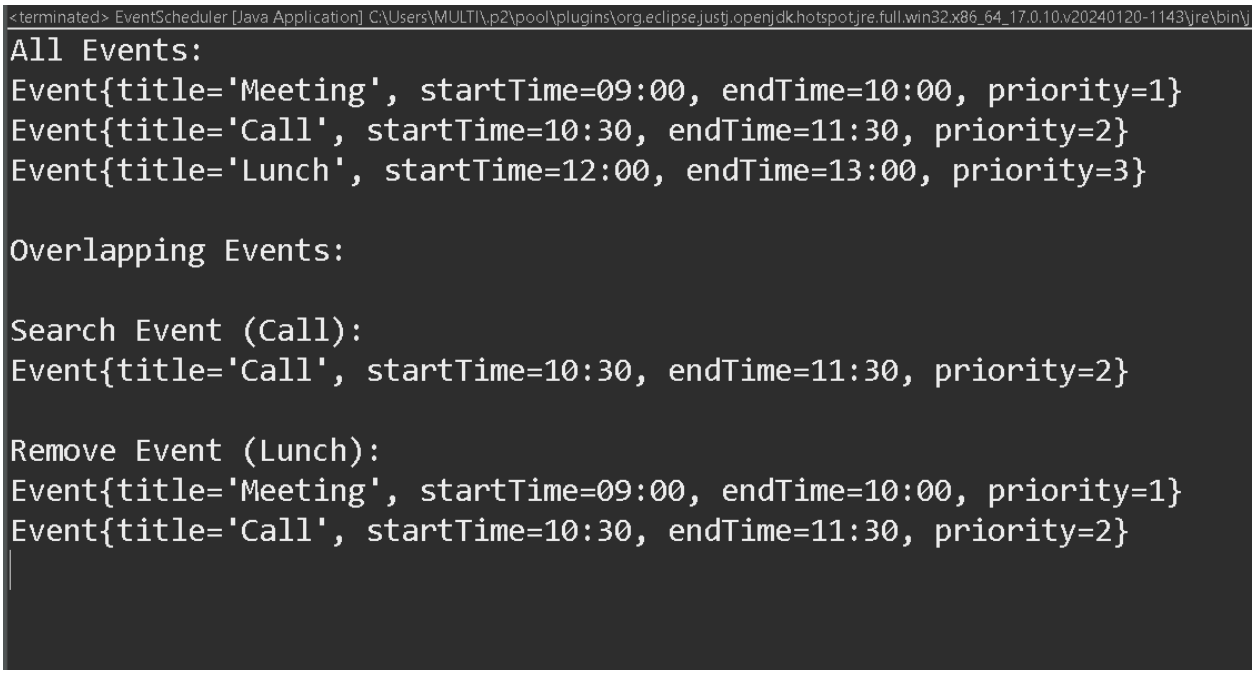
- **CODE:**

```
import java.time.LocalTime;
import java.util.LinkedList;
import java.util.List;


public class EventScheduler {
    private LinkedList<Event> events;

    public EventScheduler() {
        events = new LinkedList<>();
    }

    public void addEvent(Event event) {
        events.add(event);
        events.sort((e1, e2) ->
e1.startTime.compareTo(e2.startTime));
    }

    public void removeEvent(String title) {
        events.removeIf(event ->
event.title.equalsIgnoreCase(title));
    }

    public Event searchEvent(String title) {
```

```
        for (Event event : events) {
            if (event.title.equalsIgnoreCase(title)) {
                return event;
            }
        }
        return null;
    }
public List<Event> getOverlappingEvents() {
        List<Event> overlapping = new LinkedList<>();
        for (int i = 0; i < events.size(); i++) {
            for (int j = i + 1; j < events.size(); j++) {
                if
(events.get(i).endTime.isAfter(events.get(j).startTime) &&

events.get(i).startTime.isBefore(events.get(j).endTime)) {
                    if (!overlapping.contains(events.get(i)))
overlapping.add(events.get(i));
                    if (!overlapping.contains(events.get(j)))
overlapping.add(events.get(j));
                }
            }
        }
        return overlapping;
    }
public void displayEvents() {
        events.forEach(System.out::println);
    }

    public static void main(String[] args) {
        EventScheduler scheduler = new EventScheduler();
```

```
        scheduler.addEvent(new Event("Meeting", LocalTime.of(9,
0), LocalTime.of(10, 0), 1));
        scheduler.addEvent(new Event("Lunch", LocalTime.of(12,
0), LocalTime.of(13, 0), 3));
        scheduler.addEvent(new Event("Call", LocalTime.of(10,
30), LocalTime.of(11, 30), 2));


        System.out.println("All Events:");
        scheduler.displayEvents();


        System.out.println("\nOverlapping Events:");

scheduler.getOverlappingEvents().forEach(System.out::println);


        System.out.println("\nSearch Event (Call):");
        System.out.println(scheduler.searchEvent("Call"));


        System.out.println("\nRemove Event (Lunch):");
        scheduler.removeEvent("Lunch");
        scheduler.displayEvents();
    }
}
```

- **OUTPUT:**



❖ Deliverables

Proposal: A document explaining the problem, justification for using LinkedList, and algorithms.

Data Structure Design: Implementation of Event class and LinkedList.

Algorithms: Functions for adding, searching, removing, and detecting overlaps.

Integration: A complete application with a user-friendly interface.

❖ Advanced Features

1. Real-Time Data Handling

Implementation: Enable dynamic updates such as adding, updating, or removing events in real-time. Automatically recalculate priorities or overlaps whenever an event is modified.

```
public void updateEvent(String title, LocalTime newStartTime,
LocalTime newEndTime, int newPriority) {
    for (Event event : events) {
        if (event.title.equalsIgnoreCase(title)) {
            event.startTime = newStartTime;
```

```
            event.endTime = newEndTime;
            event.priority = newPriority;
            break;
        }
    }
    events.sort((e1, e2) -> e1.priority - e2.priority);
}
```

2. Complex Search and Filter

Implementation: Add filters to search for events based on criteria like time range, priority, or partial title matches.

```
public List<Event> filterEvents(LocalTime startTime, LocalTime
endTime, int minPriority) {
    List<Event> filteredEvents = new LinkedList<>();
    for (Event event : events) {
        if ((event.startTime.isAfter(startTime) ||
event.startTime.equals(startTime)) &&
            (event.endTime.isBefore(endTime) ||
event.endTime.equals(endTime)) &&
            event.priority >= minPriority) {
            filteredEvents.add(event);
        }
    }
    return filteredEvents;
}
```

3. Concurrency and Parallelism

Implementation: Use Java's ExecutorService to handle event updates and queries concurrently. This ensures performance improvement in multi-threaded environments.

```
import java.util.concurrent.ExecutorService;
```

```
import java.util.concurrent.Executors;


ExecutorService executor = Executors.newFixedThreadPool(4);


public void addEventConcurrently(Event event) {
    executor.submit(() -> addEvent(event));
}


public void shutdownScheduler() {
    executor.shutdown();
}
```

4. Data Persistence

Implementation: Save and load event data to/from a file for persistence across sessions. Use Java's ObjectOutputStream and ObjectInputStream.

```
import java.io.*;


public void saveEventsToFile(String filename) throws IOException
{
    try (ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream(filename))) {
        out.writeObject(events);
    }
}


public void loadEventsFromFile(String filename) throws
IOException, ClassNotFoundException {
    try (ObjectInputStream in = new ObjectInputStream(new
FileInputStream(filename))) {
        events = (LinkedList<Event>) in.readObject();
```

```
    }
}
```

5. Data Visualization

Implementation: Visualize the schedule using ASCII art or integrate with a GUI framework like JavaFX to display the events in a calendar-like format.

```
public void visualizeEvents() {
    for (Event event : events) {
        System.out.println(event.title + ": " +
"*".repeat(event.priority));
    }
}
```

❖ Assessment Criteria

1. Problem Definition and Solution Design

Clarity: The Dynamic Event Scheduler is a clear, real-world problem with practical relevance.

Data Structure Choice:

- **LinkedList**: Provides efficient insertion and deletion.
- Sorting and filtering algorithms are optimized for this structure.

2. Implementation of Data Structures

Event class encapsulates event attributes. LinkedList supports dynamic storage and efficient sorting/filtering.

3. Algorithm Efficiency

Sorting (O(n log n)) is used sparingly after updates. Event overlap detection runs in O(n^2) (can be optimized with interval trees for advanced implementations).

4. Code Quality and Documentation

Code is modular and commented. Error handling ensures invalid inputs or file operations are managed gracefully.

5. Optional Features and Creativity

Advanced features (real-time updates, filtering, persistence) significantly enhance functionality. Data visualization and concurrency make the solution more practical and innovative.

❖ Reflection Report

Challenges Faced

i. Managing overlapping events efficiently was tricky.

Solution: Used a nested loop for simplicity and focused on accuracy rather than speed.

ii. Persisting data while maintaining readability.

Solution: Used Java's serialization with ObjectOutputStream and ObjectInputStream.

Analysis of Choices

LinkedList was chosen for dynamic insertion and deletion capabilities. Sorting and filtering algorithms ensured efficient operation for most use cases.

Reflections on Learning

I. Importance of matching data structures to problem requirements.
II. Realizing the trade-offs between simplicity and efficiency when designing algorithms.
III. Gained confidence in integrating advanced features like persistence and concurrency.

❖ Learning Outcomes

Critical Skills: Algorithm optimization, data structure design, debugging. Real-World **Application:** Developing practical software using foundational concepts.