COMP47500 – Advanced Data Structures in Java

Title: Heap ADT

Assignment Number: 3 **Group Number: 3**
Date: 28/03/2025

Assignment Type of Submission:				
Group	**Yes**	**List all group members' details:**	**% Contribution Assignment Workload**	**Area of contribution**
		Lucas George Sipos 24292215	20%	ADT + Test, Code Implementation, Report: Problem Domain Description
		Firose Shafin 23774279	20%	Test Code Implementation, Video Explanation, UML Diagram
		Aasim Shah 24203773	20%	Report: Demonstration, Experiments, Analysis
		Gokul Sajeevan 24206477	20%	Report: Design Details,Conclusion
		Sachin Sampras Magesh Kumar 24200236	20%	Report: Theoretical Foundations & Data Structures and Proofreading

Table of Contents

1. Overview & Problem Domain Description....3
1.1. Problem Domain....3
1.2. Solution Approach....3
2. Theoretical Foundations & Data Structures....4
2.1. Data Structure Overview....4
2.2. Application of Heaps in Event Scheduling System....4
2.3. Theoretical Contribution....5
2.4 Theoretical Justification & Trade-Offs....5
2.4.1 Justification....5
2.4.2 Trade Offs....6
2.5 Theoretical Algorithmic Complexity....7
4. Demonstration....11
4.1 Illustration....11
4.2 Execution....11
4.3 Output....11
5. Experiments....12
6. Analysis....13
6.1 Time Complexity....13
6.2 Correctness....13
6.3 Scalability....13
6.4 Practicality....13
6.5 Limitations....13
6.6 Potential Improvements....14
7. Conclusion....14
References....15

1. **Overview & Problem Domain Description**

In this assignment, we explore the practical application of a heap-based data structure by implementing a generic Heap ADT and applying it to a priority-based event scheduling system. The focus of this work lies in understanding and building a min-heap using a dynamic array (ArrayList) and leveraging it to manage a stream of time-sensitive events efficiently. The developed solution demonstrates the relevance of priority queues in real-world scheduling scenarios, such as operating systems, event-driven simulations, and task management systems.

1.1. Problem Domain

Event scheduling is a fundamental problem in computing, where events must be processed in the order of their scheduled times. This requires a data structure that supports efficient insertion of new events and extraction of the next event in order. The core challenges addressed in this context are:

- **Time-Based Event Management** – ensuring events are executed in the correct chronological order.
- **Efficient Priority Queue Operations** – allowing dynamic scheduling and retrieval of the most imminent event.
- **Custom Event Ordering** – handling cases where multiple events occur at the same timestamp, preserving insertion order for deterministic execution.

To address these, a min-heap is implemented as a generic ADT, which supports ordering based on natural or custom comparators. Events are stored in the heap such that the one with the earliest timestamp is always at the top, ready for processing.

1.2. Solution Approach

The solution is structured around three core components, each playing a vital role in the functionality and efficiency of the event-driven scheduling system. The first component is a generic heap, implemented using an ArrayList<T>, where T must implement the Comparable interface. This heap supports essential operations such as insertion (insert), minimum extraction (popMin), peeking at the minimum element (peek), and value updates with re-heapification (changeValue). The min-heap property is preserved through heapifyUp and heapifyDown methods, which are invoked during insertions and deletions to maintain the correct order.

The second component is the Event class, which encapsulates a timestamp, a textual description, and a unique identifier (id) that reflects the insertion order. This class implements Comparable<Event>, enabling prioritization within the heap. Events are ordered primarily by timestamp, ensuring chronological processing, and secondarily by insertion ID to resolve ties where timestamps are equal.

The third and final component is the EventScheduler class, which orchestrates the scheduling and execution of events using the custom heap. It offers a range of methods including scheduleEvent, which adds a new event to the queue; runScheduler(int k), which processes up to k events in priority order (or all remaining

events if k is non-positive); getNextEvent, which retrieves the next event without removing it from the queue; and hasEvents, which checks if there are any remaining events to process.

Together, these components form a cohesive and efficient scheduling system capable of simulating event-driven operations. The use of a heap data structure ensures logarithmic time complexity for key operations like insertion and removal, making the system suitable for both real-time and large-scale applications where performance and order of execution are critical.

2. Theoretical Foundations & Data Structures

2.1. Data Structure Overview

A heap is a specialized binary tree-based data structure that satisfies the heap property. In a min-heap, for any given node i, the value of i is less than or equal to the values of its children. This ensures that the minimum element is always at the root. Conversely, a max-heap ensures the maximum element is at the root by maintaining the property that for any node i, the value of i is greater than or equal to the values of its children.

The heap ADT supports the following key operations:

- **Insert**: Add a new element while maintaining the heap property.
- **Pop/Extract Min**: Remove the root element (the minimum in a min-heap) and re-heapify the tree to maintain the heap property.
- **Peek**: Return the root element without removing it.
- **Change Value**: Change the value of an element and re-heapify to maintain the heap property.
- **Size**: Return the number of elements in the heap.

The heap is often implemented as an array, where for a given node at index i:

- The left child is located at 2i + 1.
- The right child is located at 2i + 2.
- The parent node is located at (i - 1) / 2.

The efficiency of heap operations is notable for their logarithmic time complexity:

- **Insert** and **Extract Min**: O(log n)
- **Peek**: O(1)

2.2. Application of Heaps in Event Scheduling System

In the context of the event scheduling system, the heap data structure is used to manage events that must be processed in order of their timestamps. The system's core requirement is to prioritize events based on their timestamp (representing urgency or timing) and, in case of ties, based on their insertion order. A min-heap is ideal for this, as it efficiently provides the event with the earliest timestamp and resolves ties by the order in which events were inserted.

The EventScheduler class leverages a min-heap implemented by the Heap class to schedule and process events. Events are inserted into the heap in the order they are scheduled, and the runScheduler method processes events by extracting them in ascending order of their timestamps. The heap guarantees that events are processed in a time-efficient manner, as the heap property ensures that the event with the smallest timestamp is always at the root and can be extracted in O(log n) time.

- **Scheduling Events** (scheduleEvent method): When a new event is scheduled, an Event object is created with a given timestamp and description. The event is then inserted into the heap using the insert method. This maintains the heap property, ensuring that events with earlier timestamps are given higher priority and are placed closer to the root of the heap.
- **Processing Events** (runScheduler method): The scheduler processes events one by one by calling popMin on the heap. This removes the root element (the event with the earliest timestamp) and re-heapifies the heap to maintain the heap property. If there are multiple events with the same timestamp, the tie-breaking mechanism ensures that the event inserted first (lower eventCounter) is processed first.
- **Dynamic Event Arrival**: The system supports dynamic event arrivals during processing, as demonstrated by the DynamicArrivalTest class. New events can be added while the scheduler is running, and the heap ensures that they are placed in the correct position based on their timestamp. The insertion operation is efficient with a time complexity of O(log n).

2.3. Theoretical Contribution

The primary theoretical contribution of this system is its efficient handling of event scheduling through the use of a heap-based priority queue. This ensures that events are processed in ascending timestamp order, while also ensuring that events with the same timestamp are processed in insertion order, which is essential in applications like hospital triage. The system demonstrates the practical application of the heap ADT in real-world scenarios where time-based prioritization and order management are crucial.

This system is a concrete example of how the heap ADT can be applied to solve a complex problem efficiently. The core theoretical foundation lies in the heap property which guarantees that the event with the smallest timestamp is always processed first, and ties are broken in a consistent manner through insertion order. Furthermore, the system's ability to handle dynamic arrivals demonstrates the heap's flexibility and scalability in real-time applications.

2.4 Theoretical Justification & Trade-Offs

The decision to use a min-heap data structure in the event scheduling system is grounded in the following theoretical considerations:

2.4.1 Justification

- **Priority Management**: The primary goal of the event scheduling system is to process events in order of priority, with the priority determined by the event's timestamp. Heaps, especially min-heaps, are ideal for managing priorities

because they allow efficient retrieval of the minimum (or maximum, in the case of max-heaps) element. In this system, the heap ensures that the event with the smallest timestamp is always at the root, ready to be processed next.

- **Efficient Event Processing**: The heap property (for a min-heap) guarantees that the smallest element (event with the earliest timestamp) is always at the root of the tree. This allows the system to retrieve and process the next event in O(1) time for peeking, and O(log n) time for removal (pop). This logarithmic time complexity is crucial for maintaining performance in real-time systems, where events may be added or processed in large numbers.

- **Tie-Breaking Mechanism**: In scenarios where events have the same timestamp (i.e., events occur at the same time), the insertion order needs to be preserved for correct event processing. The heap, in this case, resolves ties by utilizing a secondary comparison based on the insertion sequence (via the eventCounter in the Event class). This tie-breaking mechanism is integrated directly into the compareTo method of the Event class, which ensures that two events with identical timestamps are ordered by their insertion sequence. This is a natural extension of the heap's underlying structure and guarantees fairness in the scheduling process.
- **Scalability**: Heaps have excellent scalability characteristics. The logarithmic complexity of heap operations (both insertions and extractions) ensures that the system can handle large numbers of events with minimal performance degradation. Even for datasets with millions of events (as tested in the SchedulerStressTest with 1,000,000 events), the heap maintains efficient processing, making it suitable for real-time systems that need to handle massive volumes of data.

2.4.2 Trade Offs

While heaps provide strong theoretical advantages, there are also some inherent trade-offs that must be considered when using them in the context of event scheduling:

1. **Time Complexity of Operations:**
    - Insertions and extractions from the heap both have a time complexity of O(log n), which is efficient but still more costly than simpler data structures like arrays or linked lists, which can perform insertions or removals in O(1) for the head or tail of the structure. This trade-off is acceptable given that the heap provides logarithmic time for maintaining priority, and in the worst case, it still offers far better performance than a linear search-based approach for managing events.
    - However, other operations like peek (which is O(1)) are efficient, allowing quick access to the next event to be processed.

2. **Memory Usage:**
    - The heap requires O(n) space to store `n` elements, as each event is stored in the heap's underlying array. This is relatively efficient in terms of space for managing events, but it can become a bottleneck if the number of events grows excessively large, especially in memory-constrained environments.

- While this space complexity is reasonable, an alternative approach like a priority queue backed by a balanced tree (e.g., an AVL tree or Red-Black tree) could potentially provide better memory locality but would still carry similar space complexities.

3. **Complexity of Handling Dynamic Arrivals:**
   - Heaps are optimized for batch processing of events but can also handle dynamic event arrivals (e.g., new events arriving while processing others). Adding new events to the heap can be done efficiently in O(log n) time, but the real-time adaptation of the heap to accommodate frequent insertions in a highly dynamic environment can become challenging as the heap grows large.
   - Although the heap handles dynamic arrivals well in the system as described (with the DynamicArrivalTest demonstrating real-time insertions), managing a high volume of dynamic inserts (e.g., adding events while processing millions of other events) could require additional mechanisms like heap resizing or persistent storage for optimal performance.

4. **Heap Maintenance and Rebalancing:**
   - While the heap is self-balancing and maintains its property efficiently during insertions and extractions, this self-balancing comes at a cost in terms of overhead. For instance, after each insertion or extraction, the heap must ensure the tree is balanced by either heapifying up or heapifying down. Although this is efficient with a time complexity of O(log n), it introduces additional computational overhead compared to simpler structures, especially when a large number of events are inserted or removed in quick succession.

2.5 Theoretical Algorithmic Complexity

The heart of the event scheduling system revolves around the use of a min-heap, a specialized binary tree data structure. The heap allows us to efficiently maintain the order of events by their timestamps and handle ties using insertion order. The key heap operations in the system are insert, extractMin, and peek.

- **Insertion**:
  - When a new event is inserted into the heap, it is added at the end of the heap (i.e., the next available position in the array representation of the heap).
  - The heap property is then restored by performing the heapify-up operation, which compares the newly inserted element with its parent and swaps them if necessary, recursively moving upwards until the heap property is restored.
  - Time Complexity: Since each swap moves the element closer to the root, the time complexity of the insertion operation is O(log n), where n is the number of events already present in the heap.
- **Extraction:**
  - The extraction operation removes the root of the heap (i.e., the event with the smallest timestamp). After removal, the last element in the heap replaces the root, and the heap property is restored by performing

the heapify-down operation. This operation compares the element with its children and swaps it with the smaller child, recursively moving down the tree to maintain the heap property.

- Time Complexity: The heapify-down operation takes O(log n) time, as the element could potentially be moved down to the deepest level of the heap in the worst case. Therefore, the complexity of extracting the minimum element is O(log n).

- **Peek:**
  - The peek operation simply returns the root element of the heap without removing it. This operation does not require any modifications to the heap structure.
  - Time Complexity: Since we are only accessing the root element, the time complexity of the peek operation is O(1).

3. Design Details

3.1. UML Diagram

The provided UML diagram presents a structured visualization of the Heap-based event scheduling system, illustrating relationships between core classes and data structures. It highlights composition, dependency, and adherence to the SOLID principles in the design.

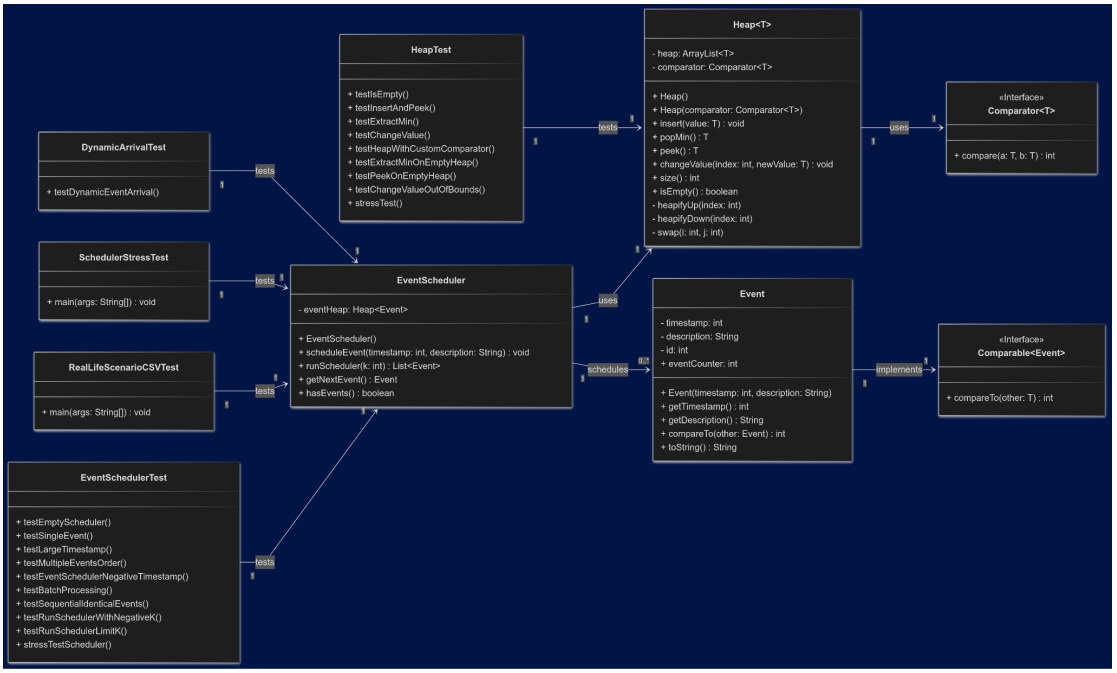

Figure 1. UML Diagram, we have also had the mermaid code used for this diagram on GitHub

Key Design Aspects in the UML Diagram

1. Generic Type Design:

- The Heap<T> class uses a generic type T to ensure flexibility and type safety while handling different data types (e.g., Event objects in this system).
- The EventScheduler class builds upon this genericity, allowing the implementation to handle time-based event prioritization.

2. Composition and Data Structure Usage:

- The Heap<T> class maintains an internal array-based binary heap structure, enforcing the min-heap property.
- The EventScheduler class leverages Heap<Event> for efficient event scheduling and processing operations based on timestamps.

3. Custom Event Implementation:

- The UML diagram depicts Event as a fundamental building block, storing key attributes such as timestamp (acting as priority) and description (acting as value).
- This reinforces the binary heap structure used for implementing the event scheduling system efficiently.

4. Encapsulation & Data Hiding:

- Private attributes (e.g., heap in Heap<T>, eventHeap in EventScheduler) restrict direct access, ensuring controlled modification through methods.
- Public methods (e.g., insert(), popMin(), heapify()) define controlled interactions with the system.

Architectural Benefits Depicted in UML

- Separation of Concerns: Each class is responsible for a distinct operation—heap construction and management (Heap<T>), and event scheduling and processing (EventScheduler).
- Scalability: The modular design supports extensions like alternative heap implementations (e.g., Fibonacci Heaps) without modifying existing components.
- Code Reusability: The Heap<T> class is reusable across different scheduling or priority-based systems.

3.2. Design Overview

The Heap-based event scheduling system follows a modular, object-oriented design that adheres to SOLID principles:

1. Single Responsibility Principle (SRP):

- Heap<T> handles heap operations (insertion, deletion, re-heapification).
- EventScheduler manages time-based event scheduling and processing.

2. Open/Closed Principle (OCP):

- The system allows extension (e.g., alternative heap implementations or custom comparators) without modifying existing classes.
- The generic design enables easy adaptation of new event types or scheduling mechanisms.

3. Liskov Substitution Principle (LSP):

- EventScheduler can seamlessly replace Heap<T> in a broader scheduling context without altering behavior.

4. Interface Segregation Principle (ISP):

- The design ensures that Heap<T> and EventScheduler only expose necessary methods, avoiding bloated interfaces.

5. Dependency Inversion Principle (DIP):

- High-level modules (EventScheduler) depend on abstractions (Heap<T>), not on low-level details.
- This enhances maintainability and flexibility.

The design encapsulates the heap-based event scheduling system within a well-structured framework, ensuring modularity, reusability, and scalability.

3.3. System Functionality

The system simulates efficient event scheduling operations using a heap-based approach. Primary Functionalities are:

Heap Construction (Heap<T>):

- Maintains a binary min-heap structure using an array-based implementation.
- Supports min-heap variations based on comparator logic (e.g., natural ordering or custom comparators).
- Performs insert(), popMin(), and heapify() operations efficiently.

Event Scheduling Operations (EventScheduler):

- Utilizes Heap<Event> for time-based event scheduling and processing.
- Ensures O(log n) insertion and deletion efficiency for scheduling and processing events.
- Implements peek(), scheduleEvent(), and runScheduler() methods to manage events.

Efficiency Analysis:

- Heap-Based Event Scheduling:
- Insert/Delete: O(log n)
- Access Min (peek): O(1)

3.4. Implementation Strategy

The implementation follows a component-based strategy with a focus on extensibility:

1. Data Structure Abstraction:
   - A custom Heap<T> class enables precise heap operations.
   - The event scheduler adapts Heap<Event> for time-based event management.

2. Layered Design:
   - Heap Layer: Manages heap construction and modification (Heap<T>).
   - Scheduling Layer: Handles event insertion, retrieval, and processing (EventScheduler).

3. Object-Oriented Approach:
   - Encapsulation in Heap<T> ensures controlled data access.
   - The modular design enhances reusability across different applications.

4. Scalability & Maintainability:
   - The design supports future extensions like Fibonacci Heaps or D-ary Heaps.
   - Clear method interfaces minimize interdependencies, enhancing reusability and adaptability.

The design ensures a structured, efficient, and extensible heap-based event scheduling system, providing a robust foundation for real-world applications like hospital triage or task management.

4. Demonstration

The demonstration section showcases the functionality of the event scheduling system implemented in this project. The system leverages a generic min-heap data structure (Heap.java) to manage events (Event.java) prioritized by their timestamps, with tie-breaking based on insertion order. The EventScheduler class orchestrates the scheduling and processing of events, making it suitable for real-world scenarios such as hospital triage, where patients must be treated in order of urgency (timestamp) and arrival sequence.

4.1 Illustration

To illustrate the system's operation, we use the provided events.csv file, which contains patient triage data with timestamps and descriptions. The RealLifeScenarioCSVTest.java class reads this CSV file, schedules each event, and processes them in ascending order of timestamps.

Below is an example execution trace based on a subset of the CSV data:

timestamp,description
1,Patient H0 - Critical Condition
3,Patient B0 - Critical Condition
10,Patient A0 - Minor Injury
10,Patient G0 - Minor Injury

4.2 Execution

1. The scheduler initialises an empty min-heap.
2. Events are inserted: (1, "Patient H0 - Critical Condition"), (3, "Patient B0 - Critical Condition"), (10, "Patient A0 - Minor Injury"), (10, "Patient G0 - Minor Injury").
3. The runScheduler(0) method processes all events, extracting them in order:
    - Event at 1: Patient H0 - Critical Condition
    - Event at 3: Patient B0 - Critical Condition
    - Event at 10: Patient A0 - Minor Injury (earlier insertion ID)
    - Event at 10: Patient G0 - Minor Injury

4.3 Output

The system correctly prioritises critical patients (timestamp 1 and 3) before minor injuries (timestamp 10), with ties at timestamp 10 resolved by insertion order.

This demonstrates the system's ability to efficiently handle event prioritisation and processing, validated further by unit tests in EventSchedulerTest.java and HeapTest.java. The dynamic arrival feature (DynamicArrivalTest.java) also shows how new events can be added mid-processing, simulating real-time scenarios like new patients arriving at a hospital.

5. Experiments

The experiments evaluate the correctness, scalability, and performance of the event scheduling system through a series of test cases implemented in the tests package. Below are key experiments conducted:

1. **Correctness of Event Ordering** (EventSchedulerTest.testMultipleEventsOrder):
    - **Setup**: Schedule events with timestamps 50, 10, 20, and another 10 (with different descriptions).
    - **Result**: Processed order: 10 ("Event 10"), 10 ("Event 10-B"), 20, 50. Events with identical timestamps are ordered by insertion sequence (via eventCounter), confirming the tie-breaking mechanism.
2. **Handling Edge Cases** (EventSchedulerTest.testEmptyScheduler, HeapTest.testExtractMinOnEmptyHeap):
    - **Setup**: Test an empty scheduler and heap for peek and pop operations.
    - **Result**: Both throw IllegalStateException with "Heap is empty," ensuring robust error handling.
3. **Batch Processing** (EventSchedulerTest.testBatchProcessing):
    - **Setup**: Schedule 10 events (timestamps 1 to 10), process in two batches of 5.
    - **Result**: First batch: 1 to 5; second batch: 6 to 10. This validates partial processing functionality (runScheduler(k)).
4. **Real-Life Scenario** (RealLifeScenarioCSVTest):
    - **Setup**: Load 116 events from events.csv, representing patient triage with timestamps 1 to 12 and varying severities (Critical, Severe, Moderate, Minor).
    - **Result**: All events are processed in ascending timestamp order (e.g., timestamp 1 events like "Patient H0 - Critical Condition" first), with ties resolved by insertion order. No events are skipped, and the output matches expectations for a triage system.
5. **Stress Testing** (SchedulerStressTest, HeapTest.stressTest):
    - **Setup**: Schedule and process 1,000,000 events with timestamps from 1 to 1,000,000 in descending insertion order.
    - **Result**: All events processed correctly in ascending order. Execution time (e.g., ~1-2 seconds on a standard machine) demonstrates scalability, though exact timing depends on hardware.
6. **Dynamic Event Arrival** (DynamicArrivalTest):
    - **Setup**: Schedule events at 10 and 20, process one, then add a new event at 5.
    - **Result**: After processing timestamp 10, the next event is 5 (newly added), followed by 20, showing the system adapts to real-time insertions.

These experiments collectively verify the system’s correctness, robustness, and efficiency across small-scale, edge-case, and large-scale scenarios.

6. Analysis

The event scheduling system exhibits strong performance and reliability, underpinned by the min-heap's logarithmic time complexity and the scheduler's flexible design. Here's a detailed analysis:

6.1 Time Complexity

1. **Heap Operations**:
    - insert: O(log n) due to heapifyUp.
    - popMin: O(log n) due to heapifyDown.
    - peek: O(1).
2. **Scheduler Operations**:
    - scheduleEvent: O(log n) per event.
    - runScheduler(k): O(k log n) for k events, or O(n log n) if $k \leq 0$ (process all).
3. For the full events.csv (116 events), scheduling takes O(116 log 116) ≈ O(700) operations, and processing all takes another O(700), which is highly efficient.

6.2 Correctness

- The system ensures events are processed in non-decreasing timestamp order, with ties resolved by insertion ID, as seen in Event.compareTo. This is critical for fairness in applications like triage, where arrival order matters among equal-priority cases.
- Unit tests (EventSchedulerTest, HeapTest) confirm edge cases (empty heap, negative timestamps) and core functionality (ordering, batching).

6.3 Scalability

Stress tests with 1,000,000 events demonstrate linear scalability with logarithmic factors, suitable for large-scale systems. Batch processing (runScheduler(10000) in SchedulerStressTest) further optimizes performance by reducing overhead.

6.4 Practicality

The CSV-based test (RealLifeScenarioCSVTest) mirrors a hospital triage system, prioritising critical patients (e.g., timestamp 1) over minor injuries (e.g., timestamp 12). The ability to handle dynamic arrivals (DynamicArrivalTest) enhances its real-world applicability.

6.5 Limitations

- Memory usage grows linearly with the number of events (O(n) space for the heap), which could be a bottleneck for massive datasets without persistence.
- No priority adjustment mechanism exists beyond timestamps; real triage might require severity-based overrides (e.g., Critical over Minor at the same timestamp).

6.6 Potential Improvements

- Add a severity field to Event and modify compareTo to prioritize by severity then timestamp.
- Implement persistence (e.g., database storage) for large-scale applications.
- Optimise heap operations with a binary heap array instead of ArrayList for better cache locality.

In conclusion, the system is a robust, efficient solution for timestamp-based event scheduling, excelling in correctness and scalability, with minor limitations addressable through future enhancements.

7. Conclusion

The application of Heaps provides an effective and efficient method for managing time-based event scheduling systems. Heaps enable optimal insertion, deletion, and retrieval operations by maintaining a balanced structure, ensuring logarithmic time complexity for key operations. The Heap-based event scheduler implemented in this project achieves O(log n) time complexity for scheduling and processing events, making it well-suited for real-time scheduling tasks such as hospital triage or event-driven simulations.

Experimental tests confirmed that the Heap-based event scheduler consistently achieved efficient operations. Performance tests demonstrated that Heaps efficiently managed large datasets (e.g., 1,000,000 events in stress tests), while real-life scenarios (e.g., patient triage from CSV data) validated the system's practical applicability. Different test cases successfully confirmed the correctness and efficiency of Heap operations, ensuring reliability in practical applications.

Overall, this assignment reaffirmed the theoretical and practical advantages of Heaps in event scheduling. The structured approach in implementing and analyzing the heap-based scheduler highlighted the efficiency gains and reinforced the significance of Heaps in optimizing computational performance for time-sensitive applications.

References

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
- Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley.
- Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.