Raymond Bautista

Southern New Hampshire University (SNHU)

CS-499: Computer Science Capstone

March 29, 2026

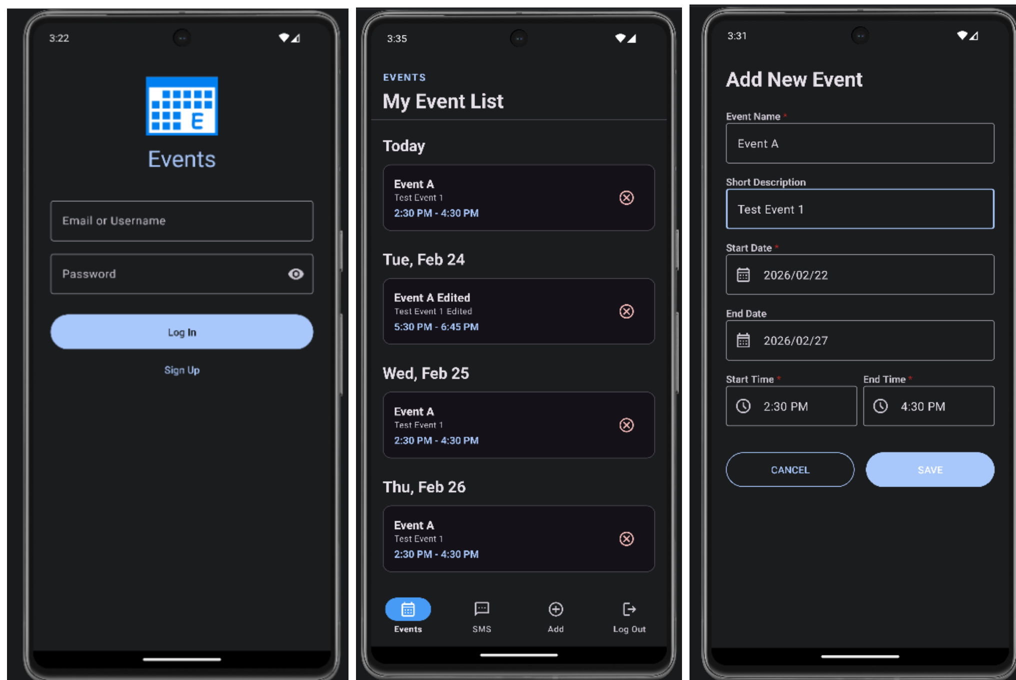
Milestone Three: Data Structures and Algorithms Enhancement

Artifact Overview

For this category, I selected the Events Android application developed in CS-360: Mobile Architecture and Programming (2026-C1). This project is a user-centered mobile application built using Android Studio, the Java programming language, and a local SQLite database. It emphasizes modern UI/UX best practices, lifecycle-aware architecture, secure authentication, and efficient performance, with the goal of reducing user cognitive load while delivering a smooth and distraction-free experience. The application functions as a personal scheduling hub, presenting events in an agenda-style list grouped by date and sorted in ascending chronological order, enabling users to easily track and manage multi-day commitments.

I selected this artifact because, in its initial implementation, it displayed events in a list grouped by date; however, users were required to manually scroll through the entire list to locate a specific event. To improve usability and efficiency, I enhanced the application by implementing a dedicated search feature that leverages pattern-matching algorithms, appropriate data structures, and sorting techniques. This functionality allows users to search for events based on keywords or matches within event titles and descriptions, returning results in ascending

chronological order. As a result, users can quickly locate, edit, or delete relevant events without navigating through the full agenda.



Proposed Enhancements

For this category, I have implemented the following improvements:

- Events search algorithm by matching pattern or keywords.
- Sorting algorithm to complement the search feature.
- Events data structure refinement to implement search and sorting algorithms.

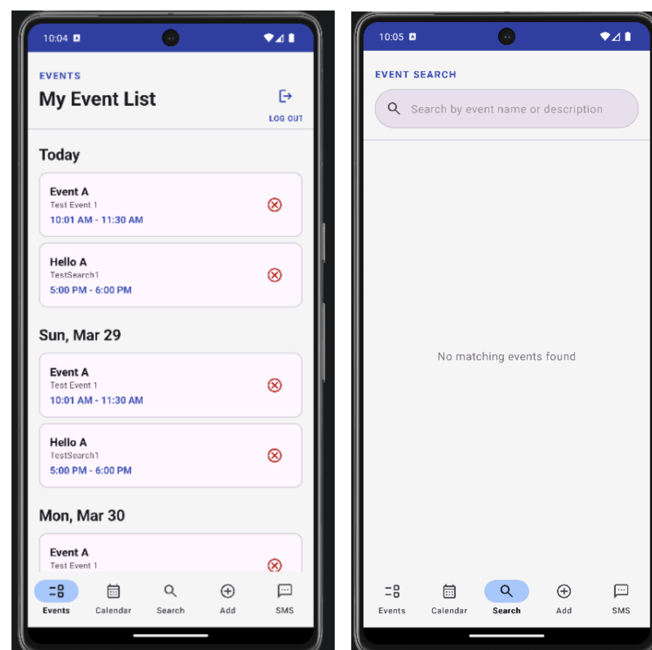
Enhancements Implementation Process and Learning Reflections

- UI Enhancements and Search Screen Design

First, I decided to isolate the search logic into a dedicated screen, similar to the previously implemented calendar feature, to prevent the MainActivity from re-rendering its

complex collapsible lists each time a user enters a character in the search bar. This design decision improves performance and maintains a smoother user experience by avoiding unnecessary UI updates during real-time input. Additionally, while the main screen retrieves event data efficiently through SQL queries via ViewModel and EventRepository, this approach is optimized for structured queries rather than flexible pattern-based searching. Therefore, for search functionality, I implemented traversal and filtering logic using algorithmic techniques in the Java programming language, enabling more dynamic and customizable search operations beyond what standard SQL queries can efficiently support.

As an initial step in integrating the feature, I added a search button with an appropriate label and icon to the bottom navigation bar. However, since the navigation bar already contained five items, I refactored the layout by relocating the logout action to a button in the top-right corner of the main screen. This decision aligns with Material Design 3 guidelines, which recommend limiting navigation bars to three to five primary destinations and ensuring that items are clearly labeled for usability and accessibility (Navigation Bar – Material Design 3, n.d.).



- Pattern Searching Algorithm Selection

To enable users to search for events by name or description in a flexible manner—returning results even when the query is incomplete or contains only partial keywords—I implemented a pattern searching approach. Pattern searching refers to a class of algorithms designed to identify sequences such as strings or words within larger data structures, allowing the system to efficiently detect relevant matches even when the input is not exact (GeeksforGeeks, 2022). There are multiple algorithms available for this purpose, each differing in terms of computational complexity, performance, and matching precision. For this application, I evaluated the following alternatives:

Naive Algorithm

This approach represents the simplest form of pattern searching. It operates by comparing the pattern against every possible position within the main string, character by character, without requiring any preprocessing phase. As a result, it does not consume additional memory beyond the input data, making it a space-efficient solution. When a match is found, the algorithm returns the index corresponding to the starting position of the pattern within the string (GeeksforGeeks, 2022). While this method is effective and straightforward for smaller datasets, its performance degrades as the size of the input string increases or when multiple patterns must be evaluated. Due to its repetitive comparisons, the time complexity can become inefficient for larger-scale applications, making it less suitable for scenarios requiring optimized search performance.

KMP Algorithm

The Knuth-Morris-Pratt (KMP) algorithm is an efficient, linear-time pattern searching technique used to identify occurrences of a pattern within a larger string. It operates by

comparing characters from left to right and is enhanced by a preprocessing step that constructs the Longest Proper Prefix which is also a Suffix (LPS) table. This table stores information about prefix–suffix matches within the pattern, allowing the algorithm to determine how far to shift the pattern when a mismatch occurs. In essence, when a mismatch is detected, the KMP algorithm leverages the information stored in the LPS table to avoid re-evaluating characters that are already known to match. This eliminates redundant comparisons and enables a linear time complexity of O(n + m), where n is the length of the input string and m is the length of the pattern. However, this performance improvement introduces a trade-off in space complexity, as the algorithm requires additional memory to store the LPS table, which has a size proportional to the pattern length (GeeksforGeeks, 2011).

Rabin-Karp Algorithm

This algorithm performs pattern searching using a rolling hash function. Unlike previously discussed algorithms, it does not rely on direct character-by-character comparison during the initial phase. Instead, it computes hash values for the pattern and substrings of the main text, allowing it to quickly filter out non-matching segments before performing detailed comparisons. The algorithm improves efficiency by calculating the hash value of the next substring window incrementally from the current hash, avoiding redundant computations. As a result, it achieves an average time complexity of O(n + m), where n is the length of the text and m is the length of the pattern. Its space complexity is O(1), as it only requires a constant amount of additional memory to store variables such as the pattern hash and the current window hash (GeeksforGeeks, 2022).

The following table presents a comparison of the time and space complexity for the evaluated pattern searching algorithms. In this context, n represents the length of the input string, while m denotes the length of the search pattern.

Algorithm	**Average Time Complexity**	**Space Complexity**	**Worst Time Complexity**
Naive	O (n*m)	O (1)	O (n^2)
KMP	O (n + m)	O (m)	O (n + m)
Rabin-Karp	O (n + m)	O (1)	O (n*m)

Algorithm Selection

Selected Search Algorithm: **KMP.**

As illustrated in the previous table, the Rabin-Karp algorithm initially appears to be an optimal choice in terms of average time and space complexity, with O(n + m) time and O(1) space. However, this approach introduces important trade-offs. It is susceptible to hash collisions, where different substrings may produce identical hash values, requiring additional string comparisons to confirm matches. These extra validations can degrade performance in the worst case. Additionally, improper handling of hashing operations may introduce risks such as modulo overflow, affecting reliability (GeeksforGeeks, 2011).

In contrast, the Knuth-Morris-Pratt (KMP) algorithm also achieves a linear time complexity of O (n + m) but provides stronger guarantees by maintaining this performance even in the worst-case scenario. Its efficiency is derived from the use of the LPS (Longest Proper Prefix which is also a Suffix) table, which enables the algorithm to avoid redundant comparisons. Specifically, the text pointer is never reset, while the pattern pointer is shifted intelligently based

on previously computed prefix information, ensuring consistent and efficient traversal (GeeksforGeeks, 2011).

Although both Rabin-Karp and KMP are suitable for this application, I selected KMP due to its deterministic behavior and robustness. Unlike Rabin-Karp, it does not rely on probabilistic hashing, eliminating the risk of collisions and ensuring predictable performance. This characteristic is particularly important in a mobile application context, where consistent response times and reliability are essential for maintaining a smooth user experience.

- Sorting Algorithm Selection

Unlike the event data displayed on the main screen and calendar, which is retrieved and sorted using SQL queries, the search feature relies on a custom algorithmic approach for data filtering. As a result, it is not appropriate to depend on SQL-based sorting for this functionality. To address this, I complemented the pattern searching algorithm with a sorting algorithm to organize the filtered results in ascending chronological order. For this purpose, I evaluated two alternative sorting algorithms to determine the most suitable option:

Quick Sort

This algorithm is based on the Divide and Conquer paradigm. It operates by selecting a pivot element and partitioning the array such that elements smaller than the pivot are placed on one side, while larger elements are placed on the other. The pivot is then positioned in its correctly sorted location. This process is recursively applied to the resulting sub-arrays until each partition contains a single element. Regarding performance, it achieves an average and best-case time complexity of $O(n \log n)$; however, its worst-case time complexity degrades to $O(n^2)$ when

the pivot selection leads to highly unbalanced partitions. In terms of space complexity, it requires O(log n) due to recursive call stack usage (Sedgewick & Wayne, 2018).

Merge Sort

This algorithm is widely recognized for its efficiency and stability. It follows the Divide and Conquer paradigm by recursively dividing the input array into two halves, sorting each sub-array independently, and then merging them to produce a fully sorted array. A key advantage of Merge Sort is that it guarantees a time complexity of O (n log n) in all cases, including the worst case, making it highly reliable for performance-critical applications. Additionally, it is a stable sorting algorithm, preserving the relative order of elements with equal values. However, this performance comes with a trade-off in space complexity, as it requires O(n) additional memory to store temporary arrays during the merging process (GeeksforGeeks, 2013).

Algorithm	**Average Time Complexity**	**Space Complexity**	**Worst Time Complexity**
Quick Sort	O (n log n)	O (n log n)	O (n^2)
Merge Sort	O (n log n)	O (n)	O (n log n)

Algorithm Selection

Selected Sorting Algorithm: **Merge Sort.**

Quick Sort is an efficient option for large datasets due to its average time complexity of O(n log n) and low memory overhead, requiring only O (log n) space for recursive calls. It is also cache-friendly, as it operates in-place without the need for auxiliary arrays. However, it has notable limitations: its worst-case time complexity degrades to O (n^2) under unfavorable pivot

selections, and it is not a stable sorting algorithm. As a result, elements with equal keys may not preserve their original relative order in the sorted output (GeeksforGeeks, 2014).

In contrast, Merge Sort requires additional memory with a space complexity of O (n) to store temporary arrays during the merging process. Despite this drawback, it offers key advantages: it is a stable sorting algorithm, preserving the relative order of equal elements, and it guarantees a consistent time complexity of O (n log n) across all cases. This predictable performance makes it particularly suitable for applications where reliability is critical. Additionally, its structure is naturally suited for parallel processing, which can be beneficial in performance-optimized environments (GeeksforGeeks, 2013).

Given the requirements of this application, specifically, sorting events in ascending chronological order where multiple events may share the same date and time, Merge Sort is the most appropriate choice due to its stability. Preserving the relative order of events ensures consistency in the user interface and avoids unintended reordering. Furthermore, its guaranteed performance contributes to smooth and responsive UI interactions, while its potential for parallelization provides opportunities for future optimization on mobile devices.

- Data Structure Selection

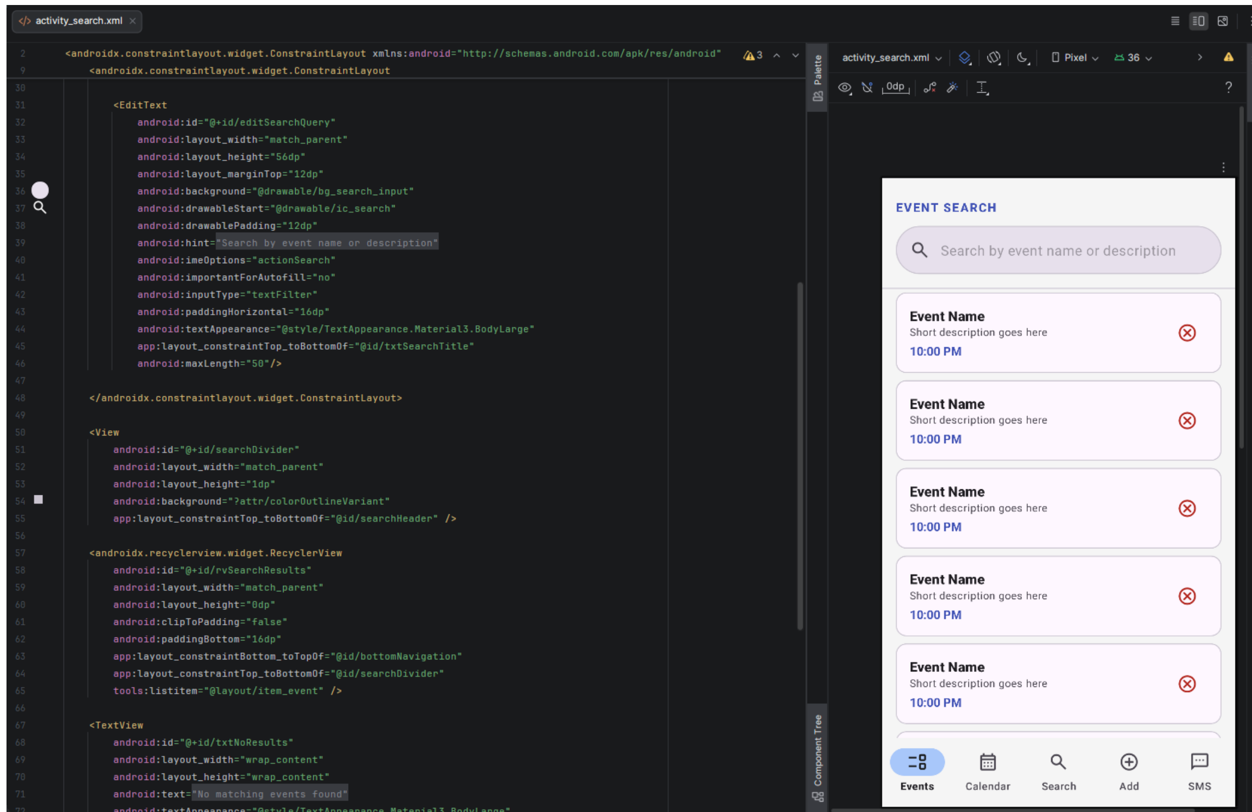
Initially, I leveraged a list (array-based structure) for this implementation, consistent with how elements were retrieved from the database. However, I encountered issues when handling deletions and UI updates, as database operations were executed faster while the UI state remained static. To address this, I enhanced the search screen implementation by synchronizing the in-memory data with the database, removing events from the local cache immediately after they are deleted. Using a list structure introduced additional inefficiencies. Deletion required first

locating the element through a linear search with O(n) time complexity, followed by a remove(index) operation, which also incurs O(n) time due to element shifting. To optimize both lookup and deletion operations, I transitioned to a hash-based data structure by storing events in a Hash Map. This approach enables average-case O(1) time complexity for both retrieval and deletion, regardless of dataset size. To preserve the logical ordering of events, I implemented a LinkedHashMap, which maintains insertion order while still providing efficient hash-based operations (Oracle, n.d.). This design ensures both high performance and consistent data presentation in the user interface.

Data Structure	**Search**	**Insertion**	**Deletion**
Array	O (n)	O (n)	O (n)
Hash Table	O (1)	O (1)	O (1)

- Code Implementation

To implement this feature, I first isolated the search logic and results list from the MainActivity by creating a dedicated screen. This design prevents the main screen from re-rendering its complex collapsible elements each time a user types in the search bar, thereby improving performance and ensuring a smoother user experience. Additionally, the search functionality retrieves and displays both past and upcoming events without distinction, allowing users to locate specific events across any time range. To support this feature, I added a search icon to the bottom navigation bar, which directs users to the dedicated search screen. On this screen, results are presented in a scrollable RecyclerView, enabling efficient navigation and interaction with the filtered event list.



For this screen, I leveraged the Model-View-ViewModel (MVVM) architecture used throughout the project to enforce separation of concerns. Within the SearchViewModel, I implemented a hash-based data structure, the Knuth-Morris-Pratt (KMP) pattern searching algorithm, and the Merge Sort algorithm to ensure efficient and scalable processing of search and sorting operations. In this design, the View layer is responsible solely for managing UI elements and user interactions, while all data processing and business logic are encapsulated within the ViewModel. Additionally, the SearchViewModel utilizes Mutable Live Data to propagate updates to the UI in real time, ensuring responsiveness as users interact with the search feature. The implementation incorporates iterative and recursive techniques, as well as divide-and-conquer strategies, to support efficient searching and sorting capabilities. On the SearchActivity, I applied the DRY (Don’t Repeat Yourself) principle by reusing the existing EventViewModel from the main screen to retrieve raw event data from the database, while delegating all search-

specific processing to the SearchViewModel. This approach promotes modularity, reduces redundancy, and maintains a clear separation between data retrieval and data processing responsibilities.

```java
// MutableLiveData list of events that updates in real time based on user search input
3 usages
private final MutableLiveData<List<Event>> searchResults = new MutableLiveData<>();

/**
 * Optimized LinkedHashMap to achieve O(1) time complexity
 * for search and delete while maintaining insertion order
 */
5 usages
private final Map<Integer, Event> eventMap = new LinkedHashMap<>();

// Stores the current search query to refresh the list if the data changes
6 usages
private String currentQuery = "";

no usages
public SearchViewModel(Application application) { super(application); }

1 usage
public LiveData<List<Event>> getSearchResults() { return searchResults; }
```

```java
/**
 * Core Search Logic
 * Filtering: O(n * m) using KMP
 * Sorting: O(n log n) using Merge Sort
 */
3 usages
public void performSearch(String query) {
    // Creates a new empty list if there are no events
    this.currentQuery = (query == null) ? "" : query.trim();
    if (currentQuery.isEmpty()) {
        searchResults.setValue(new ArrayList<>());
        return;
    }

    List<Event> filtered = new ArrayList<>();

    // Use the Map values to iterate and filter using KMP algorithm
    for (Event e : eventMap.values()) {
        if (isMatch(e.name, currentQuery) || isMatch(e.description, currentQuery)) {
            filtered.add(e);
        }
    }

    // Apply Merge Sort for chronological ascending order
    if (!filtered.isEmpty()) {
        mergeSort(filtered, left: 0, right: filtered.size() - 1);
    }

    searchResults.setValue(filtered);
}
```

```java
// Knuth-Morris-Pratt (KMP) Pattern Search Algorithm Implementation
2 usages
private boolean isMatch(String text, String pattern) {
    // Return false if input text is empty
    if (text == null) return false;

    // Normalize input text by converting to lower-case
    String t = text.toLowerCase();
    String p = pattern.toLowerCase();


    int n = t.length();
    int m = p.length();
    if (m == 0) return true;
    if (m > n) return false;    // Exit if the patter is larger than the text

    int[] lps = computeLPSArray(p);
    int i = 0; // index for text
    int j = 0; // index for pattern

    while (i < n) {
        // Move to the next character if they match
        if (p.charAt(j) == t.charAt(i)) {
            i++; j++;
        }
        // Match Found. All characters on the pattern are equal to the same length substring
        if (j == m) return true;
        else if (i < n && p.charAt(j) != t.charAt(i)) {
            if (j != 0) j = lps[j - 1]; // Use LPS to skip unnecessary comparisons
            else i++;
        }
    }
    return false;
}
```

```java
// Implement Longest Proper Prefix Array
1 usage
private int[] computeLPSArray(String pattern) {
    int[] lps = new int[pattern.length()];
    int len = 0;
    int i = 1;
    while (i < pattern.length()) {
        // If characters match, increment size of LPS and move to the next position
        if (pattern.charAt(i) == pattern.charAt(len)) {
            len++;
            lps[i] = len;
            i++;
        } else {
            // If there is a mismatch
            // Update len to the previous lps to avoid unnecessary comparisons
            if (len != 0) len = lps[len - 1];
            // If doesn't match, set lps to compare from the beginning of the pattern
            else { lps[i] = 0; i++; }
        }
    }
    return lps;
}

// Merge Sort Algorithm Implementation
3 usages
private void mergeSort(List<Event> list, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        // Divide: Recursively split the list into halves
        mergeSort(list, left, mid);
        mergeSort(list,  left: mid + 1, right);

        // Conquer: Merge the sorted halves
        merge(list, left, mid, right);
    }
}
```

```java
1 usage
private void merge(List<Event> list, int left, int mid, int right) {
    // Create temporary lists for the two halves
    List<Event> leftSide = new ArrayList<>(list.subList(left, mid + 1));
    List<Event> rightSide = new ArrayList<>(list.subList(mid + 1, right + 1));

    int i = 0, j = 0, k = left;

    // Compare elements and merge in ascending order
    while (i < leftSide.size() && j < rightSide.size()) {
        // Using String compareTo on date format "yyyy/MM/dd"
        if (leftSide.get(i).date.compareTo(rightSide.get(j).date) <= 0) {
            list.set(k++, leftSide.get(i++));
        } else {
            list.set(k++, rightSide.get(j++));
        }
    }

    // Clean up remaining elements
    while (i < leftSide.size()) list.set(k++, leftSide.get(i++));
    while (j < rightSide.size()) list.set(k++, rightSide.get(j++));
}
```

```java
// Setup ViewModels
searchViewModel = new ViewModelProvider( owner: this).get(SearchViewModel.class);
eventViewModel = new ViewModelProvider( owner: this).get(EventViewModel.class);

// Setup Event List RecyclerView
RecyclerView rv = findViewById(R.id.rvSearchResults);
rv.setLayoutManager(new LinearLayoutManager( context: this));
adapter = new EventAdapter( listener: this);
rv.setAdapter(adapter);

// Observe the raw data from DB using EventViewModel
// Pass events to SearchViewModel's internal Map
int userId = getSharedPreferences( name: "EventPrefs", MODE_PRIVATE).getInt( key: "USER_ID", defValue: -1);
eventViewModel.getEvents(userId).observe( owner: this, List<Event> events -> {
    // Refactored: We send data to the search cache instead of keeping a local list
    searchViewModel.updateRawData(events);
});
```

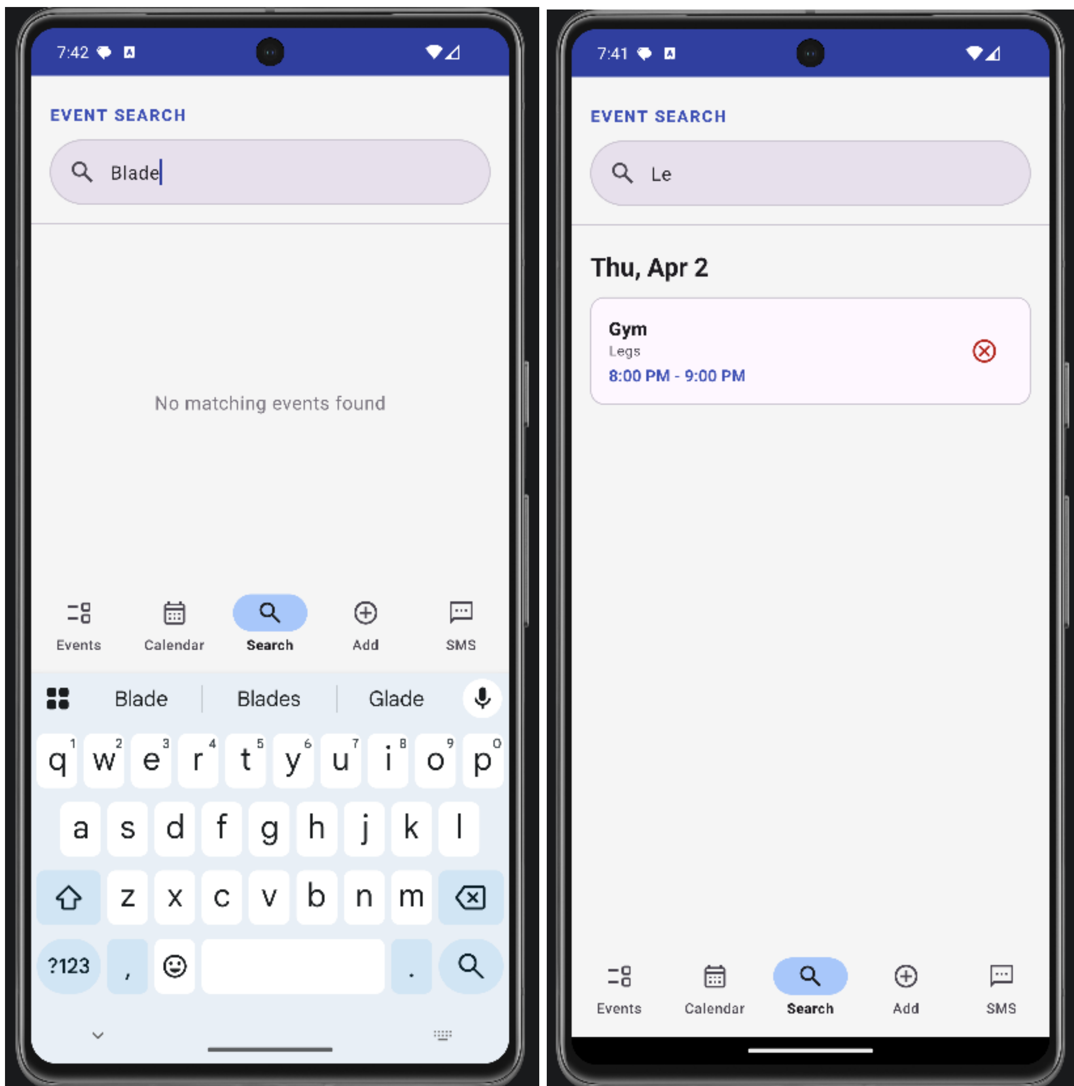
Initially, I used a list (array-based structure) to store the raw event data retrieved from the database via the EventViewModel. This approach functioned correctly for search operations using the KMP algorithm. However, during testing of the deletion feature, I observed that while events were successfully removed from the database, the cached data in the search activity remained unchanged unless a new search was executed or the user navigated to a different screen.

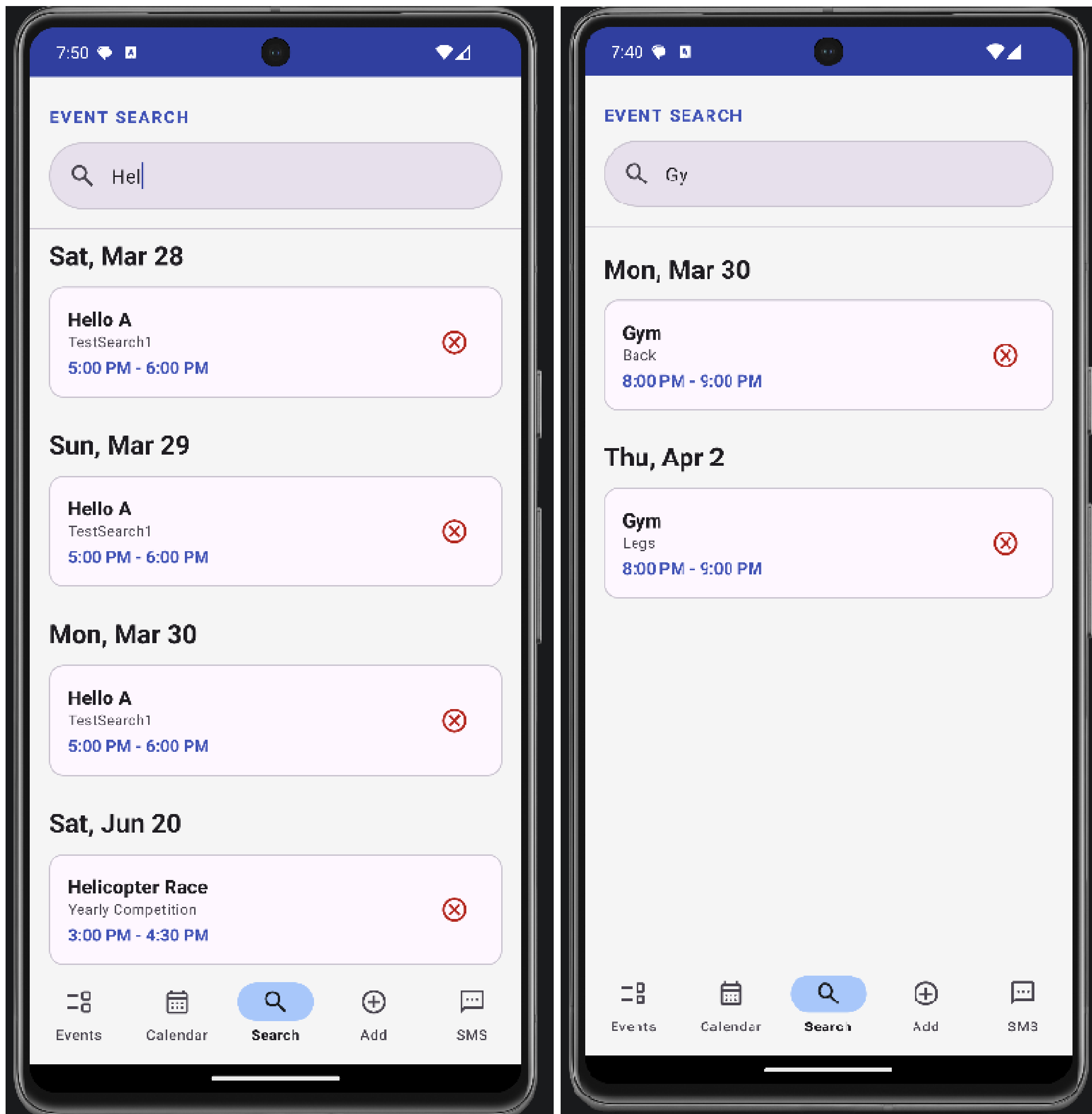
To address this limitation, I implemented an optimistic UI update strategy, which improves perceived performance by immediately reflecting changes in the user interface under the assumption that the backend operation will succeed (Optimistic State, 2025). In this case, when a user deletes an event, the system proactively updates the in-memory dataset and UI before confirming the database operation. From an architectural perspective, this logic was encapsulated within the ViewModel layer. The activity simply passes the selected event ID to the ViewModel, which handles the deletion process by coordinating with the repository and executing the appropriate SQL query. The ViewModel then updates the observable data source, allowing the UI to react automatically through LiveData. This approach ensures proper separation of concerns, maintains synchronization between the UI and data layers, and eliminates the need for the activity to manage state directly.

```
/**
 * Updates the local cache from the database observer.
 * Complexity: O(n) to populate the map.
 */
1 usage
public void updateRawData(List<Event> events) {
    eventMap.clear();
    // Insert all events from the list in the Hash Map
    if (events != null) {
        for (Event e : events) {
            eventMap.put(e.id, e);
        }
    }
    // Refresh the search results based on the new data
    performSearch(currentQuery);
}

/**
 * O(1) Optimistic Delete.
 * Removes item from local cache instantly for a smooth UI experience.
 */
1 usage
public void deleteEventOptimistically(int eventId) {
    if (eventMap.containsKey(eventId)) {
        eventMap.remove(eventId);
        performSearch(currentQuery);
    }
}
```

```java
// Allow user to delete a searched event from this screen
1 usage
@Override
public void onDelete(Event event) {
    // Delete the event from the database in a background thread
    eventViewModel.deleteEvent(event);

    // Optimistic UI Approach: Instant O(1) removal in UI (Search Cache)
    searchViewModel.deleteEventOptimistically(event.id);
}
```

- Application Tests





Conclusion and Course Outcome Alignment

Through the enhancements implemented in this category, I demonstrated the ability to design and evaluate efficient computing solutions by integrating advanced data structures and algorithms into a real-world mobile application. By selecting the Knuth-Morris-Pratt (KMP) algorithm for pattern searching, Merge Sort for stable and consistent ordering, and a hash-based data structure to optimize lookup and deletion operations, I effectively balanced performance,

scalability, and usability. These decisions required careful analysis of time and space complexity, as well as trade-offs between deterministic performance, memory usage, and user experience, directly aligning with the outcome of designing solutions using algorithmic principles and managing design trade-offs.

Additionally, I applied well-founded and innovative techniques to implement these solutions within a Model-View-ViewModel (MVVM) architecture, leveraging tools such as LiveData for real-time UI updates and adopting strategies like optimistic UI updates to enhance responsiveness. This approach not only improved system performance but also ensured a seamless and user-centered experience. Furthermore, by organizing the implementation in a modular and maintainable structure, I supported collaborative development practices and clear communication of technical concepts, aligning with the outcomes related to professional communication, effective use of computing tools, and enabling diverse audiences to understand and build upon these solutions.

References

GeeksforGeeks. (2011, April 3). KMP Algorithm for Pattern Searching. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/kmp-algorithm-for-pattern-searching/

GeeksforGeeks. (2011, May 18). *RabinKarp Algorithm for Pattern Searching*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/rabin-karp-algorithm-for-pattern-searching/

GeeksforGeeks. (2013, March 15). *Merge Sort Data Structure and Algorithms Tutorials*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/merge-sort/

GeeksforGeeks. (2014, January 7). *Quick Sort*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/quick-sort-algorithm/

GeeksforGeeks. (2022, November 8). *Introduction to Pattern Searching Data Structure and Algorithm Tutorial*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/introduction-to-pattern-searching/

Navigation bar – Material Design 3. (n.d.). Material Design. https://m3.material.io/components/navigation-bar/guidelines

Optimistic state. (2025). Flutter.dev. https://docs.flutter.dev/app-architecture/design-patterns/optimistic-state

Oracle. (n.d.). *LinkedHashMap (Java Platform SE 8)*. Docs.oracle.com. https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html

Rowell, E. (2019). *Big-O Algorithm Complexity Cheat Sheet (Know Thy Complexities!)*. Bigocheatsheet.com. https://www.bigocheatsheet.com/

Sedgewick, R., & Wayne, K. (2018). *Quicksort*. Princeton.edu. https://algs4.cs.princeton.edu/23quicksort/