UTP

UNIVERSITI TEKNOLOGI PETRONAS

TEB1113 ALGORITHM AND DATA STRUCTURE

REPORT ALGORITHM COMPARISON (ARRAY & LINKED LIST)

TTITLE: ATTENDANCE TRACKER

LECTURER NAME: DR SHASHI BHUSHAN

GROUP MEMBERS:

NAME	STUDENT ID	PROGRAMME	GITHUB REPOSITORY
LEE PEI LIN	22011261	BACHELOR OF COMPUTER SCIENCE	TEB1113_TFB2023_DSA_LAB/GroupProject_Algo at main · niliepl/TEB1113_TFB2023_DSA_LAB
HOURUN EIN BINTI ASBALAILI	22011244	BACHELOR OF COMPUTER SCIENCE	hourun-ein (Hourun Ein)
AGHISA ZAHRA	24005278	BACHELOR OF COMPUTER SCIENCE	Ghiayy
NUR ADLIN ALIYA BINTI FAKRI ZAKI	24003411	BACHELOR OF COMPUTER ENGINEERING	roboritto (adlinaliya)

Contents

Part A: Introduction

In Algorithm and Data Structure, data structures is an important part of the development of projects as it can affects the effectiveness of the algorithm. We chose to compare the data structure operations, linked list insertion vs array insertion. In this project, we explore this by implementing the data structure operations in an attendance tracker app.

Array Insertion is insertion of data in a single variable. This is easier to implement in coding as array allows multiple data insertion. Array insert data by shifting all existing element in the array to maintain order. Though it takes more resource to shift all elements, it may be useful in some cases.

Linked List Insertion is inserting data into a connected link of nodes. Each nodes have a pointer that will point to the next node until it reaches NULL signifying the end of the linked list. The pointer of a node would need to be adjusted if a new node is inserted after it. It is harder to implement in coding, but in some situation, it may be the most effective algorithm to use.

Comparing these algorithms is crucial because it helps developers choose the optimal structure based on use cases. Each of these algorithms has its own pros and cons. So, the developers can determine what they are willing to sacrifice to implement the algorithm. This project analyses their insertion algorithms experimentally and theoretically to determine their strengths and weaknesses.

Part B: Algorithm Explanation

The Attendance Tracker App allows users to check in student IDs using either an Array or Linked List data structure, enabling real-time comparison of their insertion and search performance. It visually displays operation steps, performance stats, and highlights efficiency differences between the two algorithms.

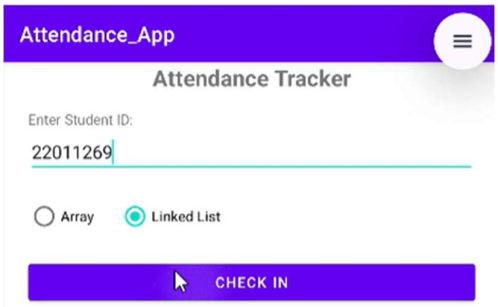
Algorithm 1: Linked List Insertion & Search

How it works?

- Insertion: inserts each new student ID at the head of the linked list. This is a fast operation because no shifting is needed.
- Search: Starts from the head node and traverses node-by-node until the target ID is found of the list ends.

Step-by-Step Example.

Insert the following example student IDs in this order:



- 22011269
- 22011265
- 22011263
- 22011261
- 22011267

Figure 2.0: Enter student IDs for linked list

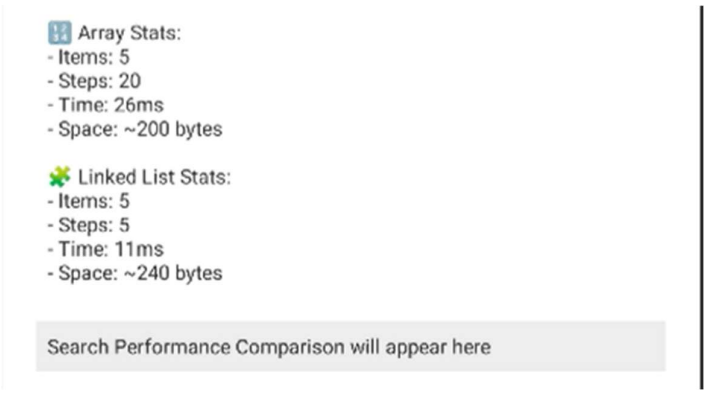


Figure 2.1: Comparison between linked list and array

Insert Coding:

```
private fun insertIntoLinkedList(id: String) {

        val start = System.nanoTime()

        val steps = StringBuilder("\n=== INSERTING INTO LINKED
LIST ===\n")



        // Remove duplicate if exists

        var prev: Node? = null

        var curr = linkedListHead

        while (curr != null) {

            if (curr.value == id) {

                steps.append("Step ${++linkedListSteps}: Found
duplicate, removing\n")

                if (prev == null) {

                    linkedListHead = curr.next

                } else {

                    prev.next = curr.next

                }

                break

            }

            prev = curr

            curr = curr.next

        }



        // Insert new node at head

        linkedListHead = Node(id, linkedListHead)
```

```
        steps.append("Step ${++linkedListSteps}: Inserted at
head\n")

        steps.append("Insert completed in ${(System.nanoTime()
- start)/1000}µs\n")



        appendToStepLog(steps.toString())

        linkedListTotalTime += System.nanoTime() - start

        showLinkedListVisual()

        updateSummary()

    }
```

Insertion:

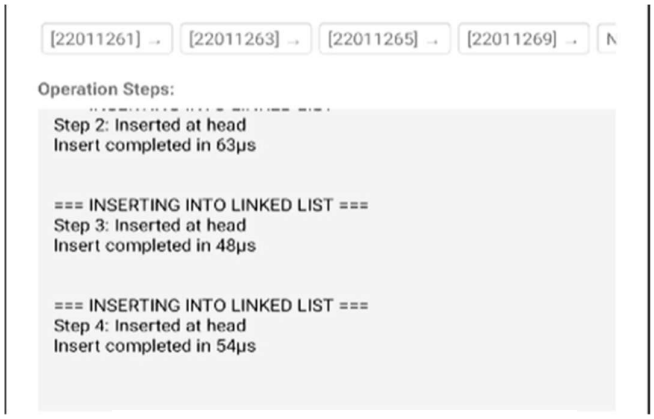


Figure 2.2: Operation steps after entering student IDs

1. Insert 22011269: becomes the head → [22011269 → NULL]
2. Insert 22011265: becomes new head → [22011265 → 22011269 → NULL]
3. Insert 22011263: becomes new head → [22011263 → 22011265 → 22011269 → NULL]
4. Insert 22011261: becomes new head → [22011261 → 22011263 → 22011265 → 22011269 → NULL]
5. Insert 22011267: becomes new head → [22011267 → 22011261 → 22011263 → 22011265 → 22011269 → NULL]

Search Coding:

```
private fun searchInLinkedList(id: String): Boolean {
        appendToStepLog("Searching linked list for '$id'...")
        val start = System.nanoTime()
        var found = false
        var position = 0
        var current = linkedListHead
        while (current != null) {
            appendToStepLog("Step ${++linkedListSteps}:
Checking position $position")
            if (current.value == id) {
                found = true
                break
            }
            current = current.next
            position++
        }
        val timeTaken = System.nanoTime() - start
        linkedListTotalTime += timeTaken

        appendToStepLog(if (found) "✅ Found at position
$position" else "❌ Not found")
        appendToStepLog("Search time: ${timeTaken/1000}µs")

        return found
    }
```

Figure 2.3: Example searching for student ID '22011269'

Searching for '22011269':

[22011267 → 22011261 → 22011263 → 22011265 → 22011269 → NULL]

- Step 1: Check 22011267 (position 0) ❌
- Step 2: Check 22011261 (position 1) ❌
- Step 3: Check 22011263 (position 2) ❌
- Step 4: Check 22011265 (position 3) ❌
- Step 5: Check 22011269 (position 4) ✅ ⇒ was found at position 4

Time taken: $3240\mu s$

Steps: 5 (linear traversal from head)

Time and Space Complexities.

Operation	Best case	Average case	Worst case	Explanation
Insertion (time)	O(1)	O(1)	O(1)	Always constant since insertion only at the beginning (no need to shift/move elements).
Search (time)	O(1)	O(n)	O(n)	Best case: item is at the head. Worst case: item is at the end or not found (have to scan all nodes).
Space	O(n)	O(n)	O(n)	One node per element (data + pointer), so space grows linearly.

Pros and Cons.

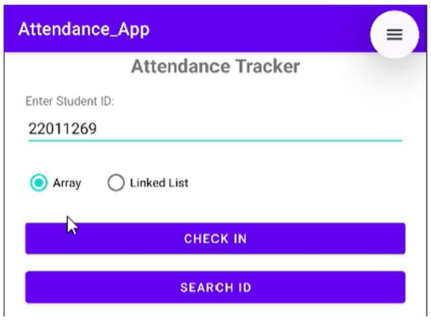
Pros	Cons
• Fast insertion at the head (constant time)	• Slower search (linear scan)
• Flexible size (no need for pre-allocation)	• Uses more memory per node (pointers)
• Good when data is always added at the front	• Cannot access by index directly

Algorithm 2: Array Insertion & Search

How it works?

- Insertion: Needs to shift all existing elements to the right before inserting at index 0.
- Search: Performs linear scan over the array to find a matching ID.

Step-by-Step Example.



Student IDs inserted: 22011269, 22011265, 22011263, 22011261, 22011267

Figure 2.4: Enter student IDs for array

Insert Coding:

```
private fun insertIntoArray(id: String) {
        val start = System.nanoTime()
        val steps = StringBuilder("\n=== INSERTING INTO ARRAY ===\n")

        // Remove duplicate if exists
        val existingIndex = arrayData.indexOf(id)
        if (existingIndex != -1) {
            arrayData.removeAt(existingIndex)
            steps.append("Step ${++arraySteps}: Removed duplicate at index $existingIndex\n")
        }

        // Make space and shift elements
```

```
        arrayData.add("")
        steps.append("Step ${++arraySteps}: Made space at
end\n")
        for (i in arrayData.size - 1 downTo 1) {
            arrayData[i] = arrayData[i - 1]
            steps.append("Step ${++arraySteps}: Shifted from
${i-1} to $i\n")
        }

        // Insert new element
        arrayData[0] = id
        steps.append("Step ${++arraySteps}: Inserted at
head\n")
        steps.append("Insert completed in ${(System.nanoTime()
- start)/1000}µs\n")

        appendToStepLog(steps.toString())
        arrayTotalTime += System.nanoTime() - start
        showArrayVisual()
        updateSummary()
    }
```

Insertion of 22011267 (at beginning):

- Step 15: Make space at end
- Step 16: Shift index 3 to 4
- Step 17: Shift index 2 to 3
- Step 18: Shift index 1 to 2
- Step 19: Shift index 0 to 1
- Step 20: Insert at index 0

Figure 2.5: Insertion for 22011267

Search Coding:

```
private fun searchInArray(id: String): Boolean {
        appendToStepLog("Searching array for '$id'...")
        val start = System.nanoTime()
        var found = false
        var position = -1

        for (i in arrayData.indices) {
            appendToStepLog("Step ${++arraySteps}: Checking
index $i")
            if (arrayData[i] == id) {
                found = true
                position = i
                break
            }
        }

        val timeTaken = System.nanoTime() - start
        arrayTotalTime += timeTaken

        appendToStepLog(if (found) "✅ Found at index
$position" else "❌ Not found")
        appendToStepLog("Search time: ${timeTaken/1000}µs")

        return found
    }
```

Searching for 22011269:

[22011267, 22011261, 22011263, 22011265, 22011269]

1. Step 23: Check index 0 ❌
2. Step 24: Check index 1 ❌
3. Step 25: Check index 2 ❌
4. Step 26: Check index 3 ❌
5. Step 27: Check index 4 ✅

Time taken: $4053\ \mu s$

Steps: 5 (linear scan)

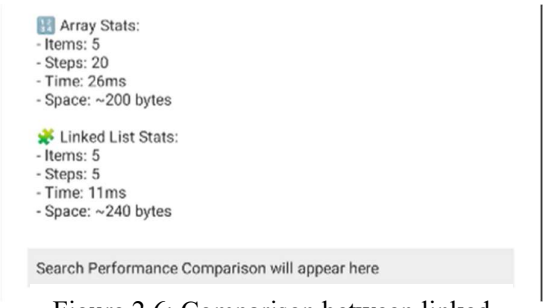


Figure 2.6: Comparison between linked list and array

Time and Space Complexities.

Operation	Best case	Average case	Worst case	Explanation
Insertion (time)	O(n)	O(n)	O(n)	Need to shift all elements to make space at index 0.
Search (time)	O(1)	O(n)	O(n)	Best case: item is at index 0. Worst case: item is at the end or not found (have to scan all nodes).
Space	O(n)	O(n)	O(n)	One slot per item.

Pros and Cons.

Pros	Cons
• Fast direct index access (useful for known position)	• Expensive insertions (need to shift)
• Contiguous memory (better cache performance)	• Fixed size unless dynamic array is implemented
	• Linear search if unsorted

Part C: Comprehensive Comparison Table

Criteria	Linked List	Array
Working Principle	Node-based (pointer-linked)	Index-based storage
Time Complexity	Access: O(n), Insert/Delete: O(1)-O(n)	Access: O(1), Insert/Delete: O(n)
Space Complexity	More (due to pointers)	Less (no pointers)
Number of Steps	Linear traversal	Linear shift on insert/delete
Best Use Cases	Frequent insert/delete, dynamic data	Frequent access, fixed data

Part D: Use Case Analysis

Algorithm 1: Linked List – Real-time insertion is needed

A linked list is highly suitable for managing music playlists where songs are frequently added, removed, or rearranged. Each song is stored in a node that links to the next (and optionally previous) song, allowing smooth navigation between tracks. This structure enables efficient insertion and deletion operations without the need to shift other elements. In music apps like Spotify, this allows users to seamlessly play the next or previous song, or update playlists on the fly, maintaining a responsive and flexible experience even as the playlist grows.

Algorithm 2: Array – Repeated access to specific index positions

Arrays are ideal for applications that require frequent access to data by position, such as a system for tracking student scores. Each student's score can be stored at a fixed index, allowing direct retrieval or update in constant time (O(1)). For example, in an educational platform where a teacher needs to view or update the score of the 15th student instantly, an array ensures fast performance. This structure works best when the number of students is fixed or changes infrequently, and when data doesn't need to be inserted or removed often.

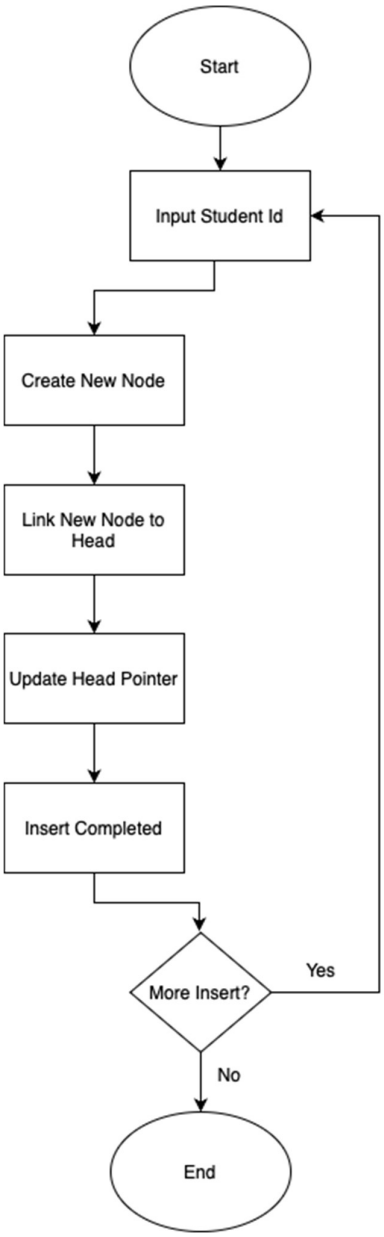
Summary

Linked lists are well-suited for dynamic, frequently modified data such as music playlists or task list, where efficient insertion, deletion, and navigation are required. Arrays, on the other hand, are preferred in scenarios where fast, position-based access is essential and the dataset remains relatively stable. The choice between the two depends on the specific operations and flexibility needed by the application.

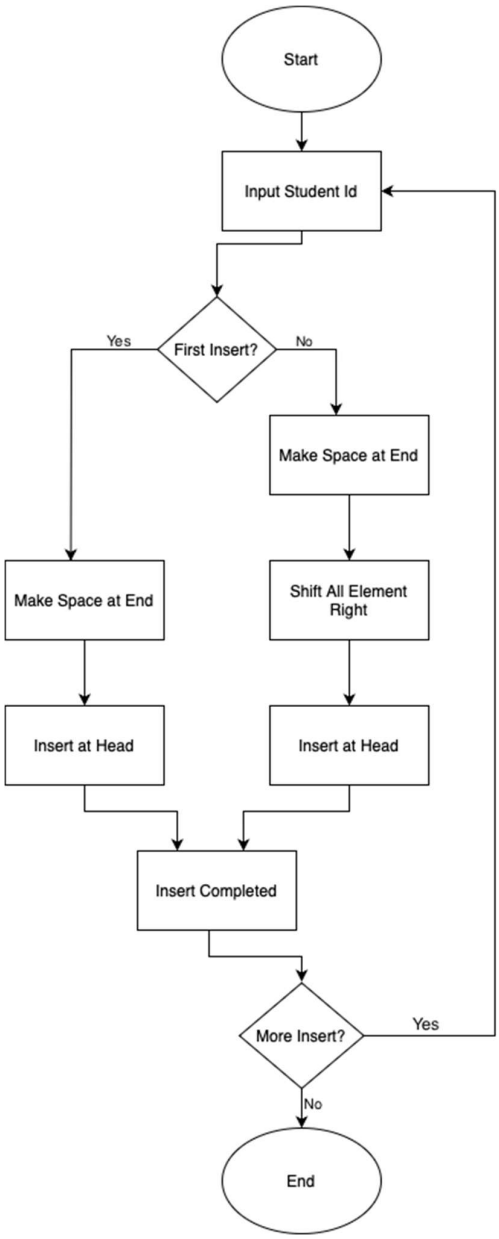
Part E: Visual Aid

- **Flowchart**

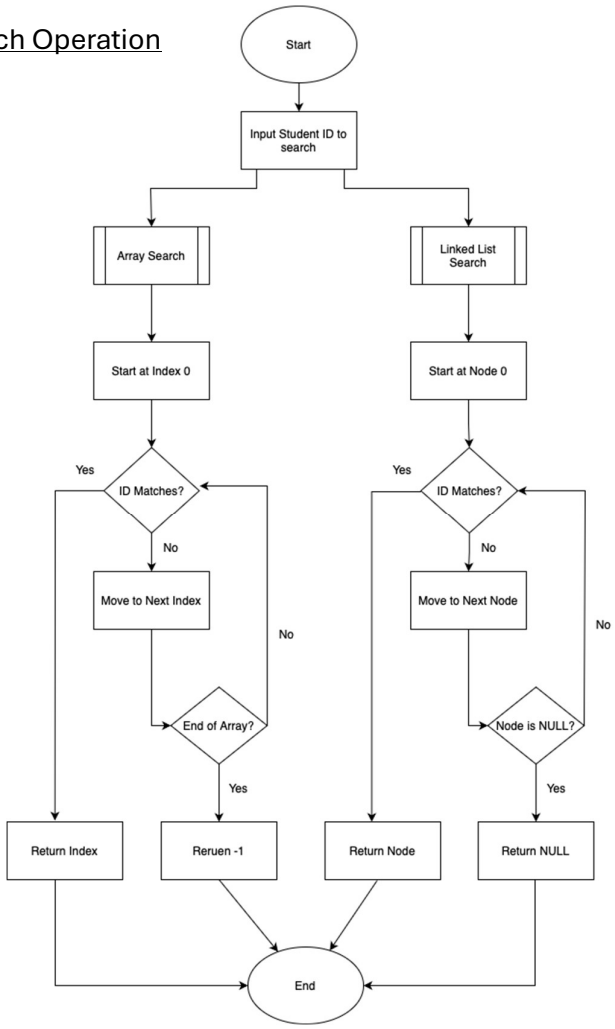
Insertion Into Array



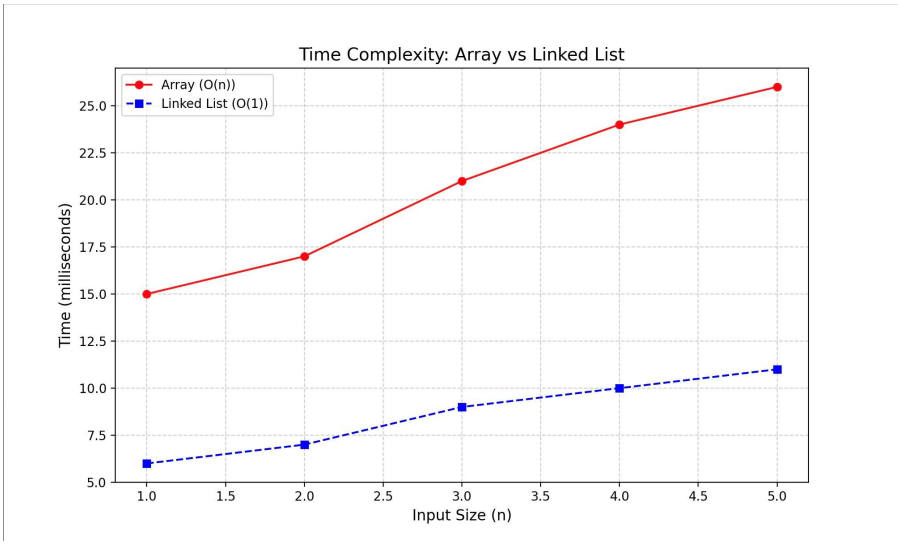
Insertion Into Linked List



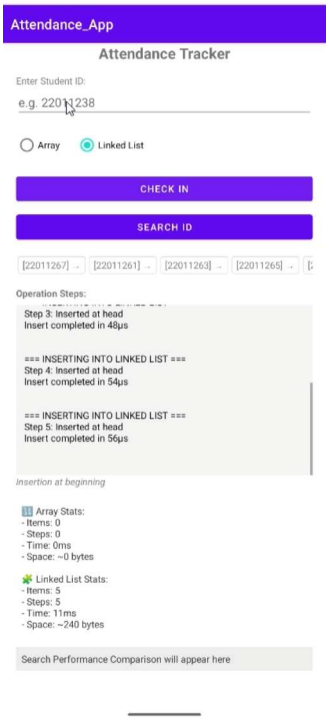
Search Operation



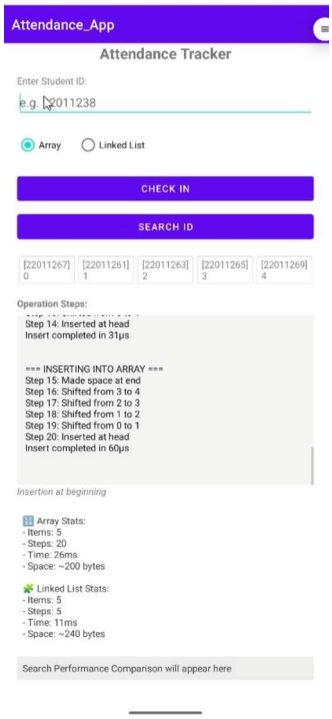
- **Operation count vs input size graphs**



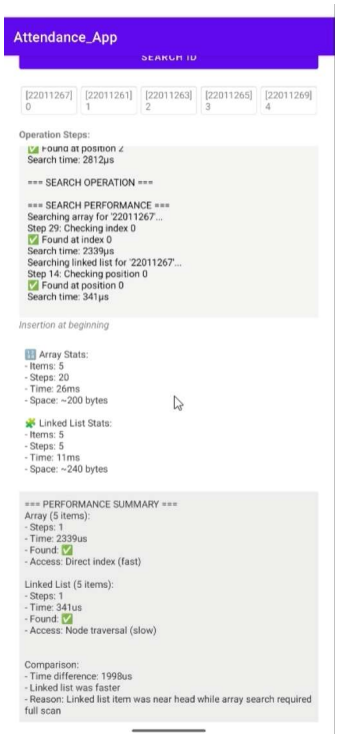
- **Screenshots from app interface**



Linked List Operation



Array Operation



Search Operation

Part F: Conclusion

As the data has shown, Linked List is the more effective algorithm as it has a lower time complexity in most of the operations. In terms of insertion performance, linked list has the average case of O(1) of time complexity when the data is inserted at the node. While array has the average case of O(n) of time complexity. This can be seen from the way linked list takes only 5 steps and 11 ms to insert 5 IDs while array took 20 steps and 26ms. In terms of search performance, both structures have O(n) for linear traversal. Though, array has its own benefit in some areas. Arrays use memory more efficiently than linked list as it lacks one other variable that linked list needs which is pointer.

Key insights from the attendance tracker are that it prioritizes insertion in which linked list is the more effective one. It sacrifices memory and search function efficiency. It depends on the developer which they are willing to sacrifices. They can use linked list if the app needs frequent insertion or deletion of data. If instead the data is pre-loaded and can be accessed by index, then array is the more effective algorithm.

Our takeaway from this project is that there are no specific algorithms that can be effective for every use case. It entirely depends on what the project demands, such as insertion or access. Hybrid solutions should be considered as a compromise between the two data structures such as dynamic arrays. This project highlights that theoretical complexities translate to the real-world applications of applications development.