JIKO (JURNAL INFORMATIKA DAN KOMPUTER)
Oktober 2025, Volume: **9**, No. **3** | Pages 496–509
doi: 10.26798/jiko.v9i3.2131
e-ISSN : 2477-3964 – p-ISSN : 2477-4413

ARTICLE

Improving Memory Efficiency on Android: Leveraging Data Structures for Optimal Performance

Muchamad Mafmudin[*,1] dan Lucia Nugraheni Harnaningrum[2]

[1]Magister Teknologi Informasi, Fakultas Teknologi Informasi, Universitas Teknologi Digital Indonesia, Yogyakarta, Indonesia
[2]Informatika, Fakultas Teknologi Informasi, Universitas Teknologi Digital Indonesia, Yogyakarta, Indonesia
*Penulis Korespondensi: correspondence@email.ac.id

(Disubmit 06-08-25; Diterima 20-08-25; Dipublikasikan online pada 20-10-25)

Abstract
This study discusses strategies for increasing memory efficiency in Android application development using optimal data structures. In the era of growing mobile device usage, especially in Indonesia, where the number of users exceeds the total population, memory management has become a significant challenge for Android developers. This study analyzes various data structures such as List, ArrayList, MutableList, and LinkedList, as well as comparisons between object and primitive data types in Kotlin. The results show that primitive data structures offer better memory efficiency and execution time than object-based data structures due to their simpler structure and lower complexity. Meanwhile, object data types like MutableList and ArrayList are more efficient for applications that require a balance between flexibility and performance, as they provide both primitive-like characteristics and useful built-in functions. This study also emphasizes the importance of understanding memory management and time complexity in optimizing Android application performance. Testing was conducted using both automated and manual methods. The findings show that Kotlin reduces memory usage by up to 2× and execution time by 28.6primitive arrays and LinkedLists showed the most stable memory performance, while MutableLists offered the best balance for object types.

KeyWords: Efficiency, Memory, Android, Structure, Data, Complexity

1. Introduction

According to a 2023 Google survey, the number of mobile device users in Indonesia reached 354 million, whereas the population of Indonesia, according to the Central Statistics Agency, is approximately 278.69 million people. This comparison shows that the number of mobile device users exceeds the population. From these data, the opportunity to develop and innovate in mobile device applications is enormous.

According to data from StatCounter (https://gs.statcounter.com/), the average Android mobile device user accounts for approximately 70% of the total mobile device users. In comparison, iOS mobile device users comprise around 30%. The above data shows that the number of Android users worldwide is significantly larger than that of iOS users. Android, with its open-source operating system base and Linux foundation, was initially designed by Android Inc. before being acquired by Google in 2005. In 2007, Android was introduced to the public, marking a significant shift in how we use technology. Although open-source, access to Google services is still provided through the Google Play Framework.

Memory management in smartphones involves multiple strategies. Efficient reference handling can reduce memory leaks[1], while the choice of data structure significantly impacts memory usage. For instance, ArrayList can outperform LinkedList in certain cases due to lower overhead, and tools such as Instruments

How to Cite: M. I. Alharits *et al.*, "Improving Memory Efficiency on Android: Leveraging Data Structures for Optimal Performance", *JIKO (JURNAL INFORMATIKA DAN KOMPUTER)*, Volume: **9**, No.**3**, Pages 496–509, Oktober 2025, doi: 10.26798/jiko.v9i3.2131.

help detect performance issues[2]. Images and multimedia also contribute heavily to memory load, where compression and formats such asWebP are recommended[1]. Effective garbage collection[3], asynchronous programming, and thread management[4],[5] are equally crucial for improving efficiency.

Optimization has been explored at various levels, from application to kernel and system services. Reinforcement learning can reduce power consumption by up to 15%[6], while PSO-LSTM achieves a 60.7reduction in page re-faults and faster launch times[7]. At the system level, selective limitation of services can save up to 150 MB of RAM[8]. Other efforts include Conditional Access for faster memory release[9], in-memory tree algorithms for hierarchical data[10], and many-to-many data structures for efficient logging and caching[11].

Additional techniques span graphics rendering (e.g., LoD optimization[12]), AI-based approaches such as SmartSplit for CNN partitioning[13], and performance surveys across applications, frameworks, and VMs[14]. Machine learning models have been applied in diverse contexts[15],[16],[17], but these do not address comparative evaluations of memory efficiency among Android data structures. Studies comparing Java and Kotlin show Kotlin's advantages in CPU, memory, and execution time efficiency[18], while optimized coding practices further improve resource usage[19].

A comprehensive survey has classified performance optimization efforts into three hierarchical levels: application, framework, and virtual machine[14]. At the application level, standard techniques include code refactoring, elimination of data redundancy, and the use of efficient data structures. At the same time, more advanced approaches have leveraged BiLSTM, Naïve Bayes with Particle Swarm Optimization, and XGBoost for domain-specific tasks[15],[16],[17]. Other studies compared Java and Kotlin in Android development, showing Kotlin's slight superiority in CPU and memory usage[18], and highlighted that improved coding practices and direct testing can further reduce resource consumption[19]. Beyond language-level improvements, strategies spanning application, kernel, and hardware layers have been proposed to enhance memory efficiency. However, despite these advances, there remains a lack of comprehensive empirical evaluations that directly compare the memory usage and execution time of different data structures—both primitive and object-based—within Android applications, a gap that is particularly critical for resource-constrained devices.

This study contributes a structured empirical comparison of data structures in Android using both memory usage and execution time as metrics, a topic that is rarely addressed in prior works. The findings can help developers optimize performance in memory-constrained environments, especially for entry- to mid-level Android devices. Kotlin shows significant advantages in memory management. Kotlin can reduce memory usage by up to 2 times compared to Java in several complex Android applications and offers features such as null safety that reduce errors that have the potential to cause memory leaks[20]. This study compares the use of data structures, including Array, List, ArrayList, MutableList, and LinkedList, to determine the optimal time and space complexity in Android application development. Efficient memory usage is a critical factor in enhancing the performance of Android applications, especially when processing large amounts of data. Empirical studies have shown that selecting appropriate data structures can significantly reduce memory overhead across different smartphone hardware architectures[21].

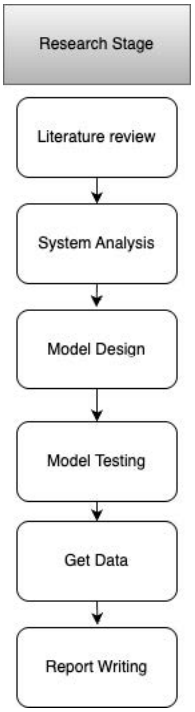
Previous studies also compared Java and Kotlin in Android development, where Kotlin was found to be slightly superior in CPU efficiency (0.65%), memory usage (up to two times more efficient), and execution time (1621 ms vs. 2268 ms). These findings, as reported in[18], support the decision to use Kotlin as the implementation language in this study. At the same time, the experimental focus remains on evaluating data structure efficiency rather than language performance. In practical terms, these results suggest that developers targeting entry- to mid-level Android devices should consider Kotlin as the preferred language to reduce memory footprint and improve responsiveness. Meanwhile, Java may still be valuable for maintaining legacy systems or projects that require compatibility with older frameworks, and in specific contexts has shown slightly better efficiency than Kotlin[22].

2. Methodology

The research stages in this study are carried out through a series of interrelated systematic steps. Each stage has an achievement indicator that functions to evaluate the progress and accuracy of the ongoing process.

The research process flows visualized in Figure 1, which starts from the literature study activity to identify previous research and relevant problems. Furthermore, a system analysis is carried out to understand the characteristics and needs of the Android application development that is the focus of the study. Based on the results of the analysis, the researcher designs a solution model that is used as a framework in simulation and testing. After that, the designed model is implemented and tested to collect performance data, both in terms of memory efficiency and time complexity. The results of this test are then analyzed and compiled into a final report that presents conclusions and recommendations based on the research findings.

This sequential step-by-step approach is designed to ensure that the development of the research model is based on a strong theoretical foundation and empirical findings. The literature study conducted at the initial stage plays an important role in comparing the various approaches that have been used in previous studies. This approach allows researchers to formulate a sharper and more contextual system analysis. Thus, the model built is not only theoretically relevant but also practically tested in Android application development simulations. The results of the model then produce accurate and representative data, so that it can be used as a basis for compiling a comprehensive and academically valid report.

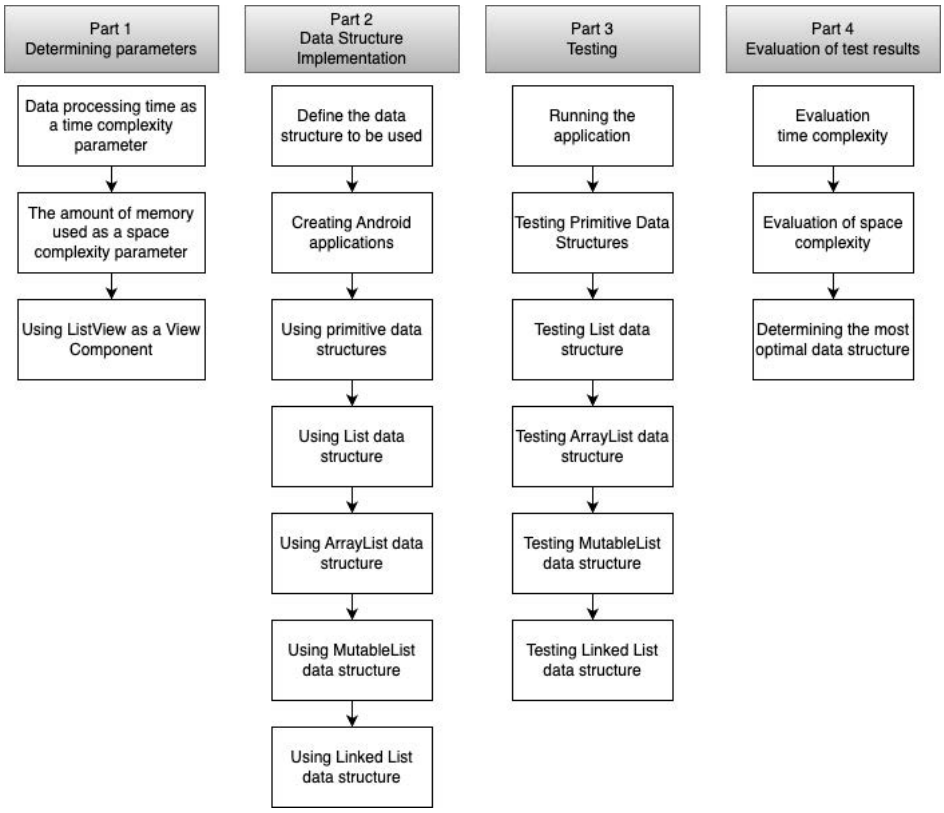


Gambar 1. Research Stage Flowchart

Literature study is used to obtain the state of the art of research, including formulating problems and proposed solutions. After conducting a literature study, the next stage is to conduct a system analysis. In the system analysis stage, the research focuses on the system used to develop Android applications. This research is conducted by observing and analyzing memory utilization in a data structure, the processing time of data structure types, and the methods used by the application to implement the data structure. After conducting a system analysis, the next stage is to create a design model for the proposed solution that will be used to provide a complete picture of the research to be carried out, including creating a smartphone memory optimization model for Android applications. The next stage is the model testing stage which begins with system development for model simulation. This stage is carried out by simulating the model applied to smartphones that have minimum and maximum specifications. This simulation is done to see the performance of the model created. The fifth stage is to obtain data and analyze the test results data.

In the fifth stage, the model created in stage three, which has been tested in stage four, is then collected based on several categories and data comparison is carried out. The final stage is to make a report and a conclusion based on the results of the previous stages.

The flowchart for the system analysis process is shown in Figure 2, which begins with determining the parameters, implementing the data structure to be used, and Testing and Test Evaluation.



Gambar 2. System Flowchart

Figure 2 depicts the detailed flow of the system analysis process, which consists of four main stages: determining parameters, implementing data structures, testing, and evaluation. Each stage is described in the following subsections.

2.1 Determining Parameters

In part one, researchers will determine the testing parameters. Commonly used parameters include Big-O notation, which is used to calculate time complexity and space complexity. Time complexity indicates how long it takes to complete an algorithm. Algorithm performance evaluations often focus on execution time and resource usage (e.g. memory) especially for real-time data processing[23]. Time complexity is calculated against the input size, while space complexity measures the memory required by the algorithm. This study shows that accounting for complexity is crucial, especially in memory-constrained systems (such as mobile devices), because space complexity affects the design of algorithms in that environment.

Meanwhile, space complexity is how much memory is needed to run the algorithm. In addition to the time parameters and memory space required, at this stage the researcher also determines the use of components that will be used in the development of the Android application. To test the use of data structures in Android applications, there are several components that can be used, and one of those used in the study is Listview. ListView is a component that displays a collection of vertically scrollable views, where each view is positioned directly below the previous view in the list[1]. ListViews that display many items can consume quite a lot of memory and CPU, potentially reducing application performance[24]. Therefore, the researcher decided to use ListView as a testing component so that it can.

2.2 Data Structure Implementation

In the second part, the researcher determines and implements several types of data structures that will be used as objects of comparison in this study. This process begins with the design of an Android application prototype that can be run directly on Android devices. This application functions as a medium for integrating and implementing various data structures, as well as a tool for testing performance and collecting empirical data.

The data structures selected for this study include Array, List, ArrayList, MutableList, and LinkedList. The selection of data structures is based on the general characteristics of each type, as well as their relevance to usage scenarios in Android applications. Each data structure has its advantages and disadvantages, especially in terms of time efficiency (time complexity) and memory usage (space complexity).

One important consideration in selecting a data structure is its impact on application performance. For example, ArrayList is one of the most frequently used data structures due to its efficienct memory allocation and direct element access via index. This structure has a relatively small memory footprint compared to LinkedList, making it more memory efficient for general use cases. However, LinkedList can offer better performance in scenarios involving intensive insertion or deletion of elements, especially at random positions in the list, although at the expense of greater memory usage due to the storage of additional pointers for each element[25].

Previous research has shown that the use of collection data structures such as List and their implementations (ArrayList, LinkedList, and MutableList) has a direct impact on application performance, both in terms of execution speed and efficiency of system resource usage. Therefore, by implementing various data structures directly into the application prototype, researchers can conduct comprehensive performance measurements based on processing time and memory allocation parameters. The results of this test then become the basis for evaluating which data structure is most optimal to use in the context of developing efficient and responsive Android applications.

An array is a data structure that stores a fixed number of values with the same type or subtype, as exemplified by several primitive data types, such as Int, String, and Float[26]. Researchers will test the use of Array by creating a List Adapter that contains arrays with primitive data types.

A List is a collection of elements that are typically sorted[26]. The List enables model customization of its elements. Researchers will use lists in research and create customization of the elements within them. An example of custom elements in the Kotlin programming language is shown in Listing 1.

Listing 1. Custom Element Example

```
data class CustomElement{
        val Id: Int,
        val name: String,
        val bmi: Float
}
```

Some functions that are often used in List are size, get, and find. In Kotlin, an ArrayList is an ordered and resizable collection of elements that utilizes an internal array for storage [26]. ArrayList is a mutable list, meaning that elements can be added, removed, and modified after they are created. ArrayList allows for custom elements to be added, similar to a List. The most commonly used functions in ArrayList are add, addAll, get, remove, find, and size.

MutableList is a generally ordered collection of elements that supports adding and removing elements [26]. In Kotlin, ArrayList is one of the implementations of the MutableList interface. Therefore, all ArrayLists are automatically Mutable Lists. However, there is a slight difference in approach. MutableList focuses on the behavior of a collection of mutable elements. Its function provides basic operations for adding, removing, and modifying elements in a list, regardless of howthe data is stored internally. ArrayList, on the other hand, is a specific implementation of MutableList that uses an internal array to store elements. This means that ArrayList offers some additional capabilities that may not be available in all other MutableList implementations. By considering the nature of MutableList as a form of interface used in ArrayList, the researcher decided to use MutableList as one of the comparison parameters in the study[27].

LinkedList is an implementation of the List and Deque interfaces. It implements all optional List operations and allows all elements (including null)[28]. LinkedList is not directly described in the kotlinlang.org documentation. However, considering that Kotlin is a programming language that runs on the JVM, it is still possible to use it, as Java programming includes the LinkedList.

2.3 Testing

In this third stage, researchers conducted performance testing on the implementation of the data created structure. This testing aims to evaluate the execution speed and memory usage of each algorithm, as both metrics are key aspects in measuring application performance[29],[30]. An official Android benchmarking library, such as Jetpack Microbenchmark, enables precise measurement of execution time in specific code segments by applying warm-up iterations, ensuring clock stability, and mitigating thermal throttling[31]. Therefore, in this study, the measurements focused on algorithm execution time and memory usage during runtime.

Testing Methods: Testing was conducted using two approaches to ensure consistent results. **Automation UI Testing**: Researchers created UI automation tests using the Android UI Automator framework. This framework allows researchers to automatically simulate user interactions with interface components (e.g., scrolling a list). With this automation, test scenarios (such as entering data or navigating a ListView) can be run repeatedly with the same pattern, making it easier to collect consistent performance data without manual errors[32]. **Manual Testing**: In addition to automation, researchers also run prototype applications manually on the Android emulator. This manual testing serves as a comparison to ensure that the application behavior and performance data obtained are in accordance following expectations. The Android emulator is used to ensure a controlled and uniform testing environment for each experiment.

The prototype interface uses a ListViewcomponent (or the equivalent LazyColumn in Jetpack Compose) to display structured data in a scrollable manner[33]. Each data structure (Array, List, ArrayList, MutableList, and LinkedList) was implemented in Kotlin within the prototype to manage the same dataset of 1,000 integer elements. Identical operations—adding, removing, and searching elements—were executed in both automated and manual tests. This execution ensured that performance differences arose from the intrinsic characteristics of the data structures rather than variations in application logic.

Performance data were recorded using the debug and profiler tools in Android Studio. **Execution Time**: Researchers inserted start and end time logs into the algorithm code using System.currentTimeMillis() to calculate duration, with results printed to Logcat[29]. **Memory Usage**: The Android Studio Profiler was employed to monitor real-time RAM consumption and to take heap snapshots for deeper inspection [30]. To ensure that hardware differences did not bias results, the tests were performed on both an Android Emulator (Pixel 7 configuration, Android 15, 2 GB RAM, 2 vCPUs). This setup provided a balance between controlled testing and realistic usage scenarios, as hardware resources such as CPU frequency and memory capacity can significantly affect performance outcomes.

Each test scenario on each data structure produces execution time data (in nanoseconds) and memory usage data (in MB) as described above. The raw data from Logcat and Memory Profiler are then further processed—averaged or compared—to be analyzed in the results and discussion sections. Thus, this testing phase ensures that the data structure implementation not only functions correctly, but also meets the expected performance aspects.

[?] In the final stage of system analysis, researchers will collect the data obtained in the previous stage for evaluation and analysis. The evaluation will be based on the parameters chosen by researchers in the first stage, namely time complexity and space complexity. First, researchers will analyze Big-O based on the data structure used, then use bar graphs to visualize the analyzed data.

3. Results and Discussion

Based on the system analysis conducted, a model was created using Kotlin as the programming language and Jetpack Compose as the tool to display the model within the Android interface. Kotlin with Jetpack Compose makes it easy to create a memory-efficient user interface. Jetpack Compose supports mo-

re efficient UI updates, resulting in reduced unnecessary memory consumption compared to traditional methods[34].

Efficient memory utilization remains a significant challenge in smartphone application development due to RAM limitations, high responsiveness requirements, and low power consumption targets. At the hardware level, techniques such as Search-in-Memory (SiM) reduce I/O overhead between storage and CPU while improving cache efficiency, achieving up to ninefold search speed increases and 45% energy saving[35]. At the application logic level, mechanisms like Conditional Access enable immediate memory release without batching, and in-memory tree construction algorithms minimize access time and join overhead for hierarchical data[9],[10]. At the data structure design level, efficient many-to-many structures support faster and more memory-conscious data logging and retrieval, particularly for use cases such as digital marketplaces and local search caching[11]. These approaches highlight that memory efficiency can be addressed across multiple layers, reinforcing the importance of selecting optimal data structures for performance-critical Android applications.

Aside from data processing and memory release strategies, graphics rendering has also seen efforts in memory and performance optimization. In the realm of graphics and rendering, the Fast Level of Detail (LoD) technique has been shown to improve frame rates in Unity-based Android applications, while also reducing both package size and memory consumption[12]. Similarly, in the context of AI-based mobile applications, efficient model partitioning has gained attention. For mobile AI development, an approach known as SmartSplit has been introduced to optimize the partitioning of convolutional neural network (CNN) models between edge devices and the cloud. This method enhances latency efficiency by up to 57and improves energy efficiency by as much as 48%[13].

The model is composed of several data samples, including object models and primitive models. In the object model, the data structures that can be implemented are List, ArrayList, MutableList, and LinkedList. In the primitive model, the data structures that can be implemented are primitive Array, List, ArrayList, MutableList, and LinkedList. In its implementation, the Array primitive cannot be implemented using the object model due to the lack of support from the programming language used.

An example implementation of calculating the load time of a data structure in the object model is shown in Listing 2.

Listing 2. Save the Start and End Time of the Data Load

```
LaunchedEffect(users) {
        renderStartTime = System.nanoTime()
        snapshotFlow { users.size() }.collect {
                renderEndTime = System.nanoTime()
        }
}
```

By using Listing 2, the start time and end time are obtained, which will determine the duration of the data loading process. The data loading time can be calculated by subtracting the end time from the start time.

Listing 3. Calculate Data Load Time

```
val renderTime by derivedStateOf {
        if (renderStartTime != null && renderEndTime != null) {
                renderEndTime!! - renderStartTime!!
        } else {
                null
        }
}
```

By using Listing 3, the data loading time is obtained in nanoseconds. After the code is implemented, data collection is carried out using UI testing. This UI testing can facilitate data collection because it allows the application to run according to the script. From the process of running UI testing, the following data is obtained.

3.1 Time Usage test Results

The results of the primitive data structure tests are presented in Table 1.

Tabel 1. Primitive(P) Data Type Test Average Data

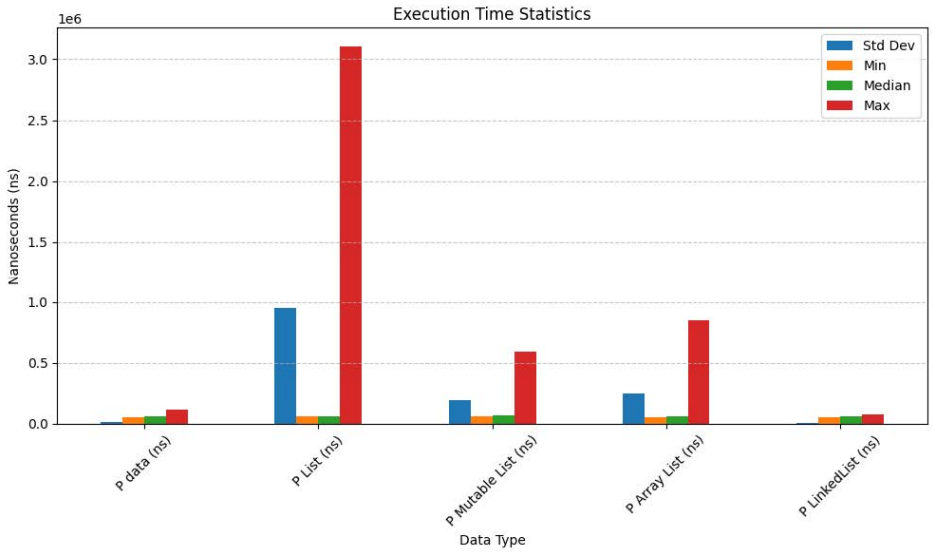
P data (ns)	P List (ns)	P Mutable List (ns)	P ArrayList (ns)	P LinkedList (ns)
72,042	59,750	58,375	59,709	5,458
64,209	73,917	428,083	850,166	60,667
56,792	185,042	131,083	66,166	64,916
11,875	60,625	75,333	56,584	56,459
63,292	3,103,292	59,583	59,666	60,291
60,583	58,667	64,708	60,416	60,667
61,416	68,167	64,041	84,958	58,958
71,208	60,625	61,291	61,042	57,958
59,834	61,292	69,792	62,125	75,542
61,000	184,292	62,625	59,666	61,791
		Average		
68,625.1	**391,566.9**	**107,491.4**	**142,049.8**	**61,470.7**

From the test results, the standard deviation, minimum, median, and maximum calculations were obtained, as shown in Table 2.

Tabel 2. Statistical Data Of Primitive (P) Data Type Testing Results

	Std Dev	Min	Median	Max
P data (ns)	17,291.42	56,792	62,354	115,875
P List (ns)	954,149.35	58,667	64,729	3,103,292
P Mutable List (ns)	190,196.68	58,375	67,250	595,583
P Array List (ns)	248,935.35	56,584	60,729	850,166
P LinkedList (ns)	5,502.82	56,459	60,479	75,542

Based on the data in Table 1 and Table 2, the following points can be analyzed. Stability: The LinkedList primitive exhibits the lowest variability (standard deviation), indicating the most consistent performance among other data structures. The List primitive exhibits high variability, displays inconsistent performance and is likely to be affected by outliers.



Gambar 3. Statistical Data of Primitive (P) Data Type Testing Results

These statistical observations provide valuable insights into the performance consistency of each primitive data structure. Building on these findings, the subsequent section presents the results of object data structure testing, as summarized in Table 3.

Median vs. Mean: For the List and MutableList primitives, the median is significantly lower than the mean, indicating a high number of outliers that affect the mean. The other data structures show medians that are close to the mean, indicating a more symmetric distribution.

Boxplot: The boxplot displays the distribution of execution times for each data structure, with the List primitive exhibiting some significant outliers. The LinkedList primitive and the data primitive show narrower and more consistent distributions.

The results of object data structure testing are presented in Table 3.

Tabel 3. Object Data Type Test Data

List (ns)	Array List (ns)	Mutable List (ns)	LinkedList (ns)
1,910,333	969,792	1,242,416	2,384,708
156,292	1,906,167	210,667	151,875
163,584	249,375	149,416	151,750
142,208	155,958	144,542	181,083
189,125	274,583	130,708	157,833
126,541	147,500	131,083	217,208
169,875	145,917	134,459	117,416
3,150,167	114,458	112,000	158,625
127,708	107,000	123,417	110,459
1,325,411	189,500	311,166	223,250
		Average	
626,837.4	426,025	268,987.4	385,420.7

From the tests carried out, data were obtained as in Table 3. With this data, the standard deviation, min, max, and median can be calculated as shown in Table 4.

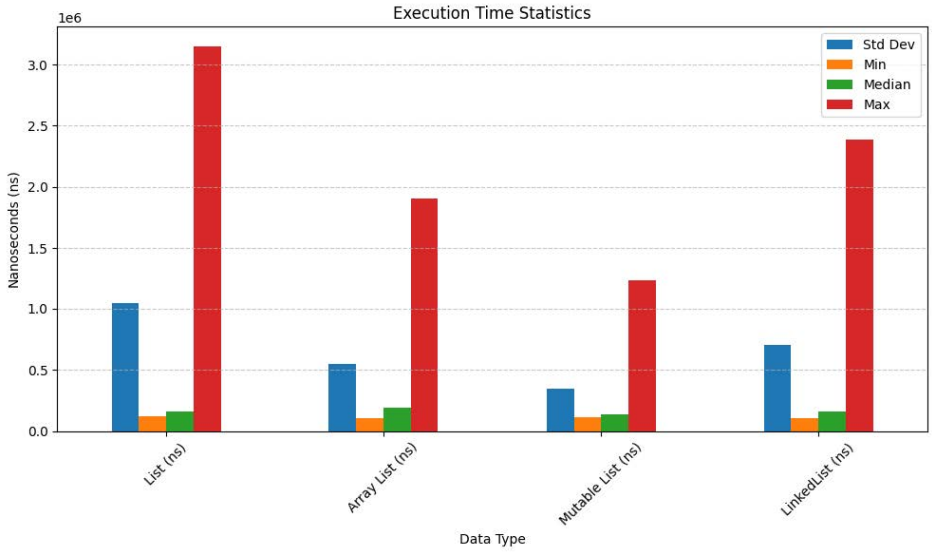
Tabel 4. Statistical Data of Object Data Type Test Results

	Std Dev	**Min**	**Median**	**Max**
List (ns)	1,045,072.99	126,541	159,938	3,150,167
Array List (ns)	549,946.42	107,000	189,500	1,906,167
Mutable List (ns)	347,147.10	112,000	139,500.5	1,242,416
LinkedList (ns)	703,428.66	110,459	158,229	2,384,708

From Table 4, the statistical metrics for each object data structure can be summarized and visualized for easier comparison. Figure 4 illustrates these results, highlighting differences in variability, minimum, median, and maximum execution times. The following points provide a detailed interpretation of each data structure.

Figure 4 presents the execution time statistics of the tested data structures. Among them, the **List** shows the highest average execution time with considerable variability, indicating significant outliers that reduce its reliability. **LinkedList** also suffers from high variability, making it less consistent despite being slightly faster than List. In contrast, **MutableList** achieves the most stable performance with lower variability, making it suitable for operations that require predictable execution time. **ArrayList** provides a balanced compromise between speed and stability, but still exhibits right-skewed distributions due to several high outliers[36].

From these results, **MutableList** emerges as the most efficient option for applications requiring consistent responsiveness, such as interactive mobile interfaces or real-time data handling. **ArrayList** is beneficial

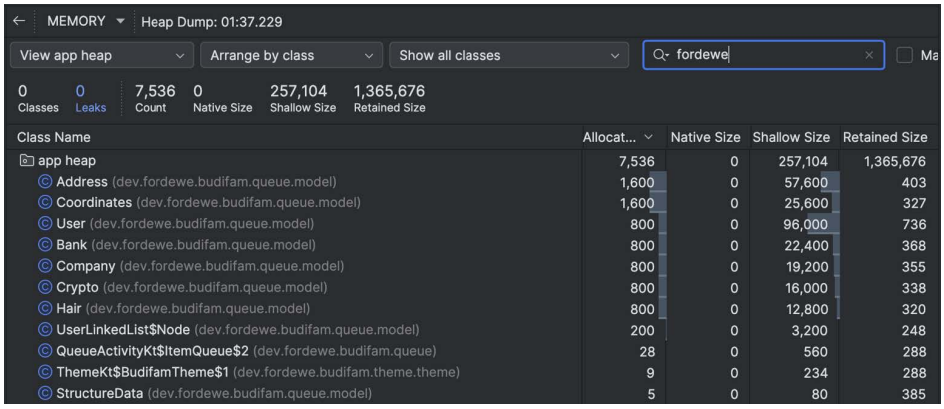


Gambar 4. Statistical Data of Primitive (P) Data Type Testing Results

in scenarios prioritizing faster average access, for example, in data retrieval operations. In contrast, **List** and **LinkedList** may not be suitable for latency-sensitive tasks due to their variability and higher memory overhead.

3.2 Memory Usage Test Results

When testing memory usage on primitive data types, the Profiler feature in Android Studio is utilized. An example of the results of capturing memory allocation is shown in the image.



Gambar 5. Memory Allocation Analysis for a LinkedList Object as Recorded by the Android Studio Profiler

Figure 5 illustrates the memory allocation structure for the LinkedList object as captured using the Android Studio Profiler. Key components shown include allocation count, shallow size, and retained size, which are used as the basis for evaluation in this study.

From the results of the memory allocation capture produced by the profiler, several components can be used as a reference, namely memory allocation, shallow size, and Retained size. Shallow size is the amount of memory used by the object itself, without counting the memory used by other objects referenced by the object. For example, for an Address object that contains a reference to a Bank object, the Shallow Size of the Address object will only include the memory used by the Address, excluding the Bank's size. Retained size is the total amount of memory that would be freed if the object and all objects that can only be accessed through the object were deleted. In other words, it includes the Shallow Size of the object itself plus the Retained Size of all objects referenced by it. Retained size gives an idea of how much memory impact an object would have if it were deleted because it includes all objects that depend on it.

The results of recording memory usage from the data structures used are presented in Table 5.

Tabel 5. Structure Data Memory Allocation Data

Data Structure	Allocation	Shallow Size	Retained Size
Object List	3703	127,840	1,341,038
Object Array List	5518	190,854	1,316,563
Object Mutable List	5496	190,362	1,274,695
Object Linked List	7536	257,104	1,365,676
primitive	44	1,476	1,234,239
primitive List	53	1,988	1,265,360
primitive Array List	61	2,480	1,194,605
primitive Mutable List	70	2,992	1,260,281
primitive Linked List	61	2,480	1,192,030

From Table 5, it can be observed that the LinkedList primitive achieved the lowest retained size among all primitive data structures (1,192,030 bytes), indicating high memory efficiency and making it suitable for applications that are sensitive to memory usage. In contrast, the MutableList primitive recorded a higher retained size (1,260,281 bytes), rendering it less optimal for scenarios with strict memory constraints.

Among object-based data structures, the MutableList object exhibited the lowest retained size (1,274,695 bytes), suggesting better memory efficiency compared to other object-based alternatives. Conversely, the LinkedList object consumed the largest retained size (1,365,676 bytes), making it the least favorable choice for memory-constrained applications.

When considering execution time, the MutableList demonstrated the most stable and predictable performance, making it preferable for scenarios requiring consistent responsiveness, such as smooth UI rendering or real-time data processing. The ArrayList offered faster average access than List and LinkedList, but its relatively high variability limits its use in latency-sensitive applications. Both List and LinkedList exhibited significant variability, reducing their suitability for operations requiring predictable performance.

Taken together, the results highlight the trade-offs between execution time and memory allocation. From the memory perspective, the LinkedList primitive proved the most efficient, while the MutableList object offered the best compromise among object-based structures. From the execution-time perspective, MutableList emerged as the most reliable, while ArrayList provided a balanced option for frequent access operations, albeit with variability concerns.

Overall, these findings indicate that no single data structure is universally optimal. Developers should select data structures according to application requirements: primitives (e.g., LinkedList or Array) for resource-constrained devices, MutableList for consistency in execution, and ArrayList for frequent-access operations with acceptable variability. These practical insights extend the contribution of this study beyond numerical evaluation by providing concrete guidance for real-world Android development.

4. Conclusion

This study demonstrates that primitive data structures consistently outperform object-based data structures in both execution time and memory efficiency. Primitive collections such as IntArray allocate significantly less memory and complete operations faster than object collections like List<Int> or LinkedList<Int>.

Empirical tests also show that LinkedList requires up to five times more memory than ArrayList, while its sequential node traversal results in slower execution and reduced stability. By comparison, array-based lists such as ArrayList provide better performance than LinkedList, but remain less efficient than primitive structures due to object boxing overhead.

These findings highlight that primitive data structures are the most suitable choice for performance-critical or large-scale Android applications. In contrast, object-based data structures may be more appropriate when flexibility and ease of development are prioritized. Developers should carefully consider this tra-

deoff when designing applications for resource-constrained devices. Future work may explore hybrid or compiler-assisted optimizations to improve memory efficiency while maintaining the usability of object collections.

Reference

Pustaka

[1] G. Developers, "Profiling memory in android," https://developer.android.com/studio/profile/memory-profiler?hl=id, 2023, [Online].

[2] A. Developer, "Data | foundation framework," https://developer.apple.com/documentation/foundation/data, 2023, [Online].

[3] Oracle, "Garbage collection in java platform, standard edition," https://docs.oracle.com/javase/, 2023, [Online].

[4] Microsoft, "Asynchronous programming and multithreading in .net," https://docs.microsoft.com/, 2023, [Online].

[5] J. Bloch, *Effective Java*, 3rd ed. Addison-Wesley, 2018.

[6] Z. Chen, T. Wang, K. Zhang, Y. Guo, and Z. Shao, "A q-learning-based display energy optimization scheme for android systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 44, no. 4, pp. 1262–1275, Apr. 2025, [Online].

[7] S. Zhao, J. Wang, S. Yu, and W. Wang, "An adaptive android memory management based on a lightweight pso-lstm model," in *2024 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, Apr. 2024, pp. 1–6, [Online].

[8] C. Piao, X. Ding, J. He, S. Jang, and M. Liu, "Implementation of image transmission based on vehicle-to-vehicle communication," *Journal of Information Processing Systems*, vol. 18, no. 2, pp. 258–267, 2022.

[9] A. Singh, T. Brown, and M. Spear, "Efficient hardware primitives for immediate memory reclamation in optimistic data structures," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023, pp. 112–120, [Online].

[10] R. Kanathur, R. H. Bhagirath, A. R., and S. A., "In-memory depth-first tree construction of hierarchical data in a rdbms," in *Proceedings of the 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS)*, 2023, pp. 1–8, [Online].

[11] T. Kuwabara, "New data structure for many-to-many relations to reduce data size, recording time, and search time," in *Proceedings of the 9th International Congress on Advanced Applied Informatics (IIAI-AAI)*, 2020, pp. 351–356, [Online].

[12] C. Zhang, Q. Liu, and Y. Wang, "Research on unity scene optimization based on fast lod technique: Performance comparison on android mobile platform," *Journal of Physics: Conference Series*, vol. 2254, no. 1, p. 012021, 2022, [Online].

[13] Ramadass and D. Murugan, "Smartsplit: Latency-energy-memory optimisation for cnn splitting on smartphone environment," *Future Generation Computer Systems*, vol. 126, pp. 153–167, 2022, [Online].

[14] M. Hort, M. Kechagia, F. Sarro, and M. Harman, "A survey of performance optimization for mobile applications," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2879–2904, Aug. 2022, [Online].

[15] A. Shajari, H. Asadi, S. Alsanwy, S. Nahavandi, and C. P. Lim, "Application of a bilstm model for detecting driver distraction caused by hand-held mobile phones, utilizing physiological signals and head motion data," in *2024 IEEE International Systems Conference (SysCon)*, vol. 22, no. 4. IEEE, Apr. 2024, pp. 1–8, [Online].

[16] A. H. Nasution and A. P. U. Siahaan, "Performance optimization of naïve bayes algorithm for malware detection on android operating systems with particle swarm optimization," *Journal of King Saud University - Computer and Information Sciences*, 2021.

[17] R. Sivakumar and K. Ramesh, "Forecasting the prices using machine learning techniques: Special reference to used mobile phones," in *Procedia Computer Science*, 2020, vol. 172, pp. 949–956, [Online].

[18] N. S. Sibarani, G. Munawar, and B. Wisnuadhi, "Analisis performa aplikasi android pada bahasa pemrograman java dan kotlin," Bandung, 2018.

[19] P. A. Fitriani, "Optimisasi kinerja aplikasi android melalui pengelolaan memori dan pengoptimalan kode," *Jurnal Dunia Data*, vol. 1, no. 1, pp. 1–22, 2024, [Online].

[20] M. Shafique, M. A. Iqbal, and R. U. Rehman, "A comparative study of kotlin and java for android development," *Journal of Computer Applications*, vol. 42, no. 2, pp. 45–50, 2020.

[21] L. N. Harnaningrum, P. D. W. Anggoro, A. H. Nasyuha, M. Mafmudin, and A. Wijaya, "Optimizing memory usage in android smartphones: A comparative analysis of data structures across different hardware architectures," *International Journal of Interactive Mobile Technologies (iJIM)*, vol. 19, no. 15, pp. 97–109, 2025, [Online].

[22] P. Gajek and M. Plechawska-Wójcik, "Performance comparison of the java and kotlin programming languages based on an auto-scroller mobile game," *Journal of Computer Sciences Institute*, vol. 33, pp. 285–291, 2024, [Online].

[23] A. Shabbir, A. Majeed, M. Iftikhar, and R. H. Ali, "A review of algorithms' complexities on different valued sorted and unsorted data," in *2023 International Conference on Innovative Computing (ICIC)*. IEEE, 2023, pp. 216–224.

[24] R. R. Kolle and R. M. S., "Enhancing the performance of android," *International Journal for Research in Applied Science and Engineering Technology (IJRASET)*, vol. 10, no. 6, pp. 110–118, 2022.

[25] D. Costa, A. Andrzejak, J. Sebők, and D. Lo, "Empirical study of usage and performance of java collections," in *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. ACM, 2017, pp. 403–414.

[26] JetBrains, "List in kotlin," https://kotlinlang.org/docs/collections.html#list, 2023, [Online].

[27] M. A. Saca, *Refactoring Improving the Design of Existing Code*. Addison-Wesley, 2017, vol. 2018-Janua.

[28] G. Developers, "Reducing app size by compressing images," https://developer.android.com/studio/write/resource-mgmt.html, 2023, [Online].

[29] M. D. Shah, "Lib metamorphosis: A performance analysis framework for exchanging data structures in performance sensitive applications," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 379–389.

[30] S. Yoon and Y. Choi, "Performance profiling of mobile applications using android studio profiler," *Journal of Information Processing Systems*, vol. 17, no. 1, pp. 73–81, 2021.

[31] A. Developers, "Jetpack microbenchmark overview," https://developer.android.com/topic/performance/benchmarking/microbenchmark-overview, 2024, [Online; accessed 2025-08-18].

[32] E. Morais, V. Camargo, and M. T. Valente, "On the use of android ui automator for automated user interface testing," *Journal of Systems and Software*, vol. 143, pp. 351–364, 2018.

[33] S. Hasan, M. Ahmed, F. Islam, and R. Kabir, "Performance comparison between android listview and recyclerview," in *2017 IEEE Region 10 Humanitarian Technology Conference (R10-HTC)*. IEEE, 2017, pp. 681–686.

[34] A. A. Donovan and B. W. Kernighan, *The Go Programming Language*. Upper Saddle River, NJ: Addison-Wesley, 2020.

[35] Y.-C. Chen, Y.-H. Chang, and T.-W. Kuo, "Search-in-memory: Reliable, versatile, and efficient data matching in ssd's nand flash memory chip for data indexing acceleration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 11, pp. 3864–3876, 2024, [Online].

[36] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 2nd ed. Addison-Wesley Professional, 2018, [Online].