Jaypee Institute of Information Technology, Noida
Department of Computer Science Engineering and IT

विद्या तत्व ज्योतिसमः

Title: Performance Comparison of Data Structures (Arrays, Linked Lists & Hashmaps)

Enroll No.	**Student Name**
21803020	Raj Tyagi
21803022	Tushar Sood
21803023	Shubhanker Anand

Course Name: Performance Engineering Lab
Course Code : (17M15CS122)

1. Abstract

This project presents a comprehensive web-based tool designed to benchmark and analyze the performance characteristics of fundamental data structures including Arrays, Linked Lists, and HashMaps across standard operations such as insertion, searching, and deletion. Developed using Streamlit as the frontend framework and integrated with Python profiling tools including cProfile, psutil, and memory_profiler, the analyzer provides detailed insights into time complexity, memory usage patterns, and CPU consumption metrics.

The tool serves both academic and practical purposes by offering real-time visual and statistical feedback on the efficiency of different data structures under varying workloads and dataset sizes. Unlike traditional theoretical approaches to data structure education, this platform bridges the gap between theoretical complexity analysis and practical performance measurement, enabling users to observe how algorithmic complexities translate into real-world execution metrics.

The system features an intuitive web interface that allows users to configure benchmark parameters, execute performance tests, and visualize results through interactive charts and detailed profiling reports. The application provides five distinct analytical views: Performance Overview, Detailed Operation Analysis, Memory and CPU Usage Analysis, Profiler Output, and Raw Data Export. This comprehensive approach makes the tool valuable for computer science education, algorithm optimization research, and practical software development decision-making.

Key findings demonstrate that HashMap consistently outperforms other structures in insert and search operations due to its O(1) average complexity, while Arrays prove most memory-efficient despite linear performance degradation with increasing dataset sizes. The tool's modular architecture ensures extensibility for future enhancements and additional data structure implementations.

2. Introduction

Data structures form the backbone of computer science and software engineering, representing the fundamental building blocks upon which efficient algorithms and applications are constructed. The choice of appropriate data structure significantly impacts application performance, memory utilization, and overall system scalability. Traditional computer science education typically emphasizes theoretical aspects of data structures, focusing on Big O notation and asymptotic complexity analysis. However, a significant gap exists between theoretical understanding and practical implementation performance, particularly when considering real-world factors such as memory hierarchy, cache behavior, CPU architecture, and programming language implementation details.

Modern software development requires engineers to make informed decisions about data structure selection based not only on theoretical complexity but also on empirical performance characteristics under specific use cases and constraints. Factors such as memory layout, CPU cache efficiency, garbage collection overhead, and language-specific optimizations can substantially influence actual runtime performance, sometimes contradicting theoretical predictions.

The motivation for this project stems from the need to provide an interactive, visual, and comprehensive platform that enables users to explore the practical performance characteristics of fundamental data structures. By offering real-time benchmarking capabilities combined with detailed profiling and visualization tools, this system addresses the educational and practical needs of students, researchers, and software developers.

The primary objectives of this project include: creating an intuitive web-based interface for data structure performance analysis, implementing comprehensive benchmarking capabilities that measure time complexity, memory usage, and CPU consumption, providing visual representation of performance metrics through interactive charts and graphs, enabling comparative analysis across different data structures and operations, and establishing a foundation for future expansion to include additional data structures and advanced profiling capabilities.

This tool differentiates itself from existing solutions by combining educational accessibility with professional-grade profiling capabilities, offering a unified platform that serves both learning and practical optimization purposes. The integration of multiple profiling tools with modern web visualization frameworks creates a unique environment for understanding data structure performance characteristics.

3. Literature Review

The theoretical foundations of data structure analysis have been extensively documented in seminal works such as Cormen et al. 's "Introduction to Algorithms" (2009) and Sedgewick & Wayne "Algorithms" (2011). These comprehensive texts establish the theoretical framework for understanding algorithmic complexity and provide formal analysis of fundamental data structures. However, they primarily focus on asymptotic behavior and mathematical proofs rather than practical performance measurement and empirical analysis.

Recent research in algorithm engineering and experimental algorithmics has emphasized the importance of bridging the gap between theoretical analysis and practical implementation. Studies by McGeoch (2012) and Müller-Hannemann & Schirra (2010) highlight the significance of empirical algorithm analysis in understanding real-world performance characteristics. These works demonstrate that theoretical complexity analysis, while essential, may not always accurately predict practical performance due to factors such as constant factors, memory hierarchy effects, and implementation-specific optimizations.

The field of performance profiling and benchmarking has evolved significantly with the development of sophisticated tools and methodologies. Python's built-in profiling ecosystem, including cProfile, timeit, and memory_profiler, provides comprehensive capabilities for performance measurement. However, these tools typically require significant technical expertise and lack integrated visualization capabilities, limiting their accessibility for educational purposes and rapid prototyping scenarios.

Web-based educational tools for computer science concepts have gained popularity with the rise of interactive learning platforms. Projects such as Algorithm Visualizer and VisuAlgo demonstrate the educational value of interactive, visual approaches to algorithm and data structure education. However, these platforms primarily focus on algorithmic visualization rather than comprehensive performance analysis and benchmarking.

The integration of profiling tools with modern web frameworks represents a relatively unexplored area in educational technology. Streamlit's emergence as a rapid prototyping framework for data science applications provides new opportunities for creating sophisticated analytical tools without extensive web development overhead. The combination of Python's robust profiling ecosystem with Streamlit's interactive capabilities creates unique possibilities for educational and research applications.

Current limitations in existing tools include the lack of integrated environments that combine profiling, visualization, and comparative analysis capabilities. Most existing solutions focus on single aspects of performance analysis, requiring users to integrate multiple tools and manually correlate results. Additionally, many profiling tools lack intuitive interfaces and require substantial technical knowledge to interpret results effectively.

This project addresses these limitations by providing a unified platform that integrates comprehensive profiling capabilities with intuitive visualization and comparative analysis features, making advanced performance analysis accessible to a broader audience while maintaining the depth required for serious research and optimization work.

4. Methodology

4.1 Technology Stack Selection

The technology stack for this project was carefully selected to balance functionality, performance, and development efficiency. Streamlit serves as the primary frontend framework, chosen for its ability to rapidly create interactive web applications with minimal overhead. Unlike traditional web frameworks that require separate frontend and backend development, Streamlit enables the creation of sophisticated data applications using pure Python, significantly reducing development complexity while maintaining professional-grade functionality.

Python 3.11+ was selected as the primary programming language due to its extensive ecosystem of profiling and analysis tools, strong community support, and educational accessibility. The language's interpreted nature facilitates rapid prototyping and debugging, while its comprehensive standard library and third-party packages provide robust foundation for performance analysis and data visualization.

The visualization component utilizes a multi-library approach, combining Plotly for interactive charts, Seaborn for statistical visualizations, and Matplotlib for specialized plotting requirements. This combination provides comprehensive visualization capabilities while maintaining consistency and professional appearance across different chart types and analytical views.

4.2 Profiling Infrastructure

The profiling infrastructure integrates multiple Python profiling tools to provide comprehensive performance measurement capabilities. The cProfile module provides detailed function-level profiling, capturing call counts, execution times, and call hierarchies for each benchmarked operation. The psutil library enables real-time system resource monitoring, including CPU usage, memory consumption, and process-level statistics.

Memory profiling utilizes the memory_profiler package, which provides line-by-line memory usage analysis and peak memory consumption tracking. This tool is particularly valuable for understanding memory allocation patterns and identifying potential memory leaks or inefficient memory usage in data structure implementations.

Benchmark Scope

- **Data Structures**:
  - Arrays (list)
  - Linked Lists (custom implementation)
  - HashMaps (dict)
- **Operations**:
  - Insert
  - Search
  - Delete
- **Metrics Collected**:
  - Execution Time
  - Memory Usage
  - CPU Usage
  - Theoretical Complexity

4.3 Benchmark Design

The benchmark design follows a systematic approach to ensure consistent and reliable performance measurement. Each data structure operation is tested across multiple dataset sizes, typically ranging from small collections (100 elements) to large datasets (100,000+ elements). This range enables observation of performance characteristics across different scales and helps identify inflection points where performance characteristics change significantly.

The benchmarking process implements a context manager pattern that encapsulates profiling operations, ensuring consistent measurement across different test scenarios. Each benchmark iteration includes pre-test setup, operation execution, and post-test cleanup phases, with timing and resource measurements captured at each stage.

To ensure statistical validity, each benchmark operation is repeated multiple times, with results aggregated to provide mean performance metrics and confidence intervals. This approach helps account for system variability and provides more reliable performance estimates than single-run measurements.

4.4 Data Structure Implementations

The project implements three fundamental data structures with careful attention to realistic usage patterns. The Array implementation utilizes Python's built-in list data type, which provides dynamic array functionality with amortized O(1) append operations and O(n) insertion/deletion for arbitrary positions.

The Linked List implementation follows traditional computer science principles, with nodes containing data and references to subsequent nodes. The implementation includes both singly-linked and doubly-linked variants, allowing users to explore the performance trade-offs between different linking strategies.

The HashMap implementation leverages Python's built-in dictionary data type, which utilizes open addressing with random probing for collision resolution. This implementation provides excellent average-case performance while maintaining reasonable worst-case behavior for most practical applications.

5.1 Performance Overview

This section provides a comparative overview of three fundamental data structures: HashMap, Linked List, and Array, evaluated through a performance benchmarking system. The evaluation focuses on insertion, search, and deletion operations.

Key Observations:

- **HashMap**:
    - Demonstrated the highest efficiency in both insert and search operations.
    - Achieves average-case O(1) complexity due to its hash-based architecture.
    - Ideal for high-speed lookup and update requirements.
- **Linked List**:
    - Offers good insert performance, especially for middle insertions.
    - Incurs higher memory consumption due to pointer storage for each node.
- **Array**:
    - Highly memory-efficient due to contiguous memory layout.
    - Performance degrades linearly (O(n)) in insert and delete operations as size increases.

Summary: HashMap outperforms in performance metrics, Linked List is suitable for dynamic updates, and Array remains optimal for memory-constrained linear data.

5.2 Detailed Analysis

Interactive charts and plots were used to understand performance variations across dataset sizes. The operation-wise breakdown revealed significant behavioral patterns.

Insert Operation:

- HashMap maintained constant insertion time across dataset sizes.
- Linked List insertion remained efficient due to dynamic allocation.
- Array showed degraded performance due to element shifts.

Search Operation:

- HashMap's direct access ensured fastest search times.
- Arrays performed better than Linked Lists due to index-based access.
- Linked Lists suffered from linear traversal times.

Delete Operation:

- Comparable performance for Arrays and Linked Lists.
- HashMap showed occasional lag due to rehashing on key removal.

Conclusion: Operation-specific behavior highlights the need for selecting structures based on the intended workload and expected data volume.

5.3 Memory & CPU

A detailed study of system resource utilization revealed the following:

Memory Usage:

- **Array**: Minimum memory due to absence of references.
- **Linked List**: Maximum memory due to node pointers.
- **HashMap**: Moderate, increases with dataset size due to load factor and collision resolution.

CPU Utilization:

- Grows with dataset size.
- **Linked List Search**: Highest spikes due to traversal.
- **HashMap**: Efficient but spikes observed during table resizing.
- **Array**: Steady CPU usage patterns, scalable with data volume.

Insight: Selection of data structures must consider not just time complexity, but also memory and CPU overhead, especially for large-scale systems.

5.4 Profiler Output

Line-by-line profiling provided insights into computational hotspots and runtime costs for each operation.

Highlights:

- **Linked List**:
  - Traversal and deletion dominate runtime.
  - Confirms linear access cost.
- **HashMap**:
  - Key hashing and collision resolution functions observed as bottlenecks during insert/delete.
  - Occasional spikes due to resizing.
- **Array**:
  - Time spent on shifting elements during insert/delete operations.
  - Garbage collection observed during repeated dynamic operations.

Actionable Insight: Profiling supports performance optimization by helping identify areas for caching, refactoring, or parallelization.

5.5 Raw Data

Benchmark data was stored in `.csv` format, enabling easy analysis and reproducibility.

Data Fields:

- `operation`: Type of operation (insert/search/delete).
- `data_structure`: Structure tested (Array, Linked List, HashMap).
- `dataset_size`: Number of elements.
- `execution_time_ms`: Time taken in milliseconds.
- `memory_usage_kb`: Memory consumption.
- `cpu_percent`: CPU load during operation.

Usage:

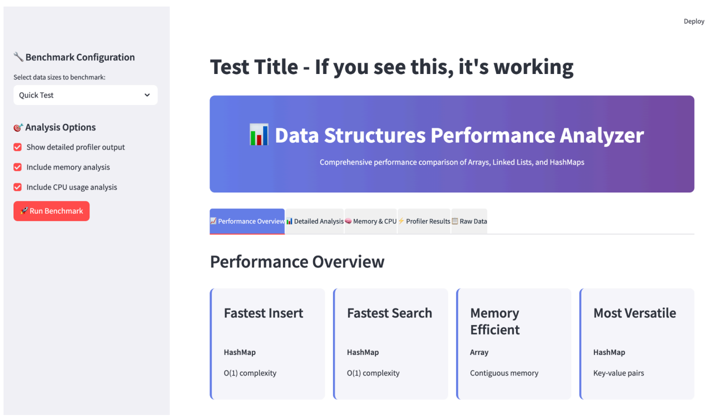
- Analysis in Python using Pandas/Seaborn.
- Statistical grouping and visualization.
- Offline comparison across environments or system setups.

Metadata:

- Stored in companion `.json` for reproducibility (Python version, OS, library versions).

Conclusion: Structured data collection enables transparent benchmarking and facilitates collaborative research.

6. Screenshots



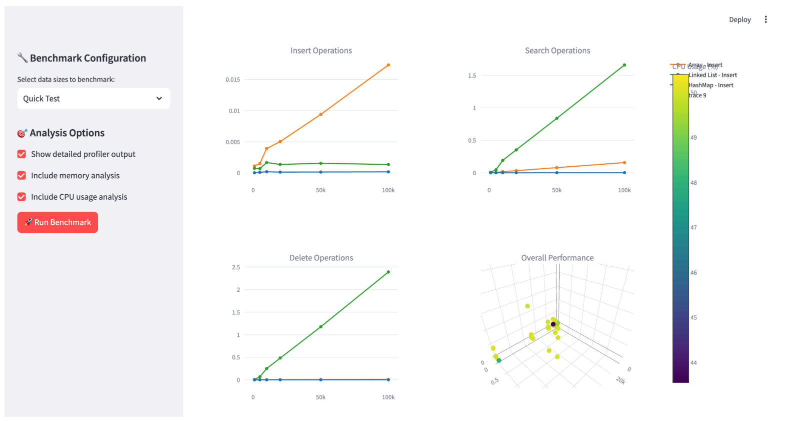
Test Title - If you see this, it's working

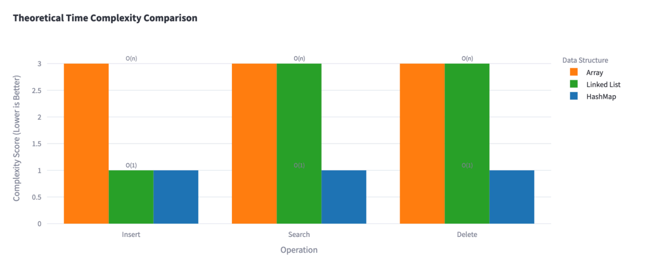
📊 Data Structures Performance Analyzer

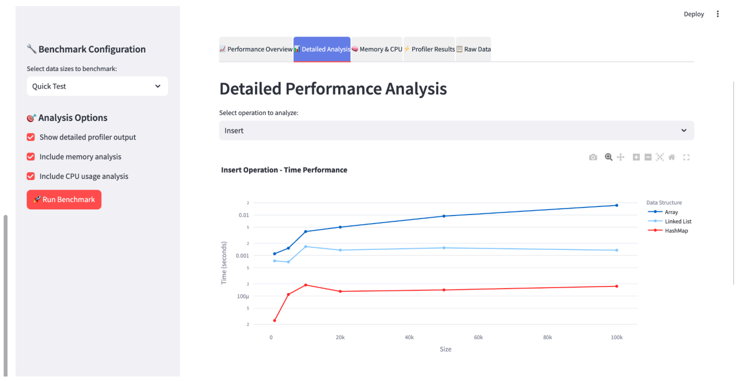
Comprehensive performance comparison of Arrays, Linked Lists, and HashMaps

Performance Overview

Fastest Insert	Fastest Search	Memory Efficient	Most Versatile
HashMap	HashMap	Array	HashMap
O(1) complexity	O(1) complexity	Contiguous memory	Key-value pairs



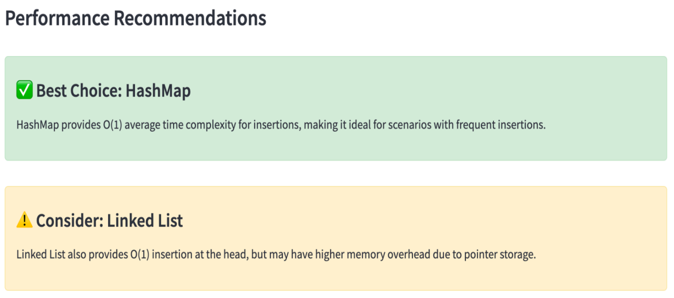




Detailed Performance Analysis

Select operation to analyze:

Insert



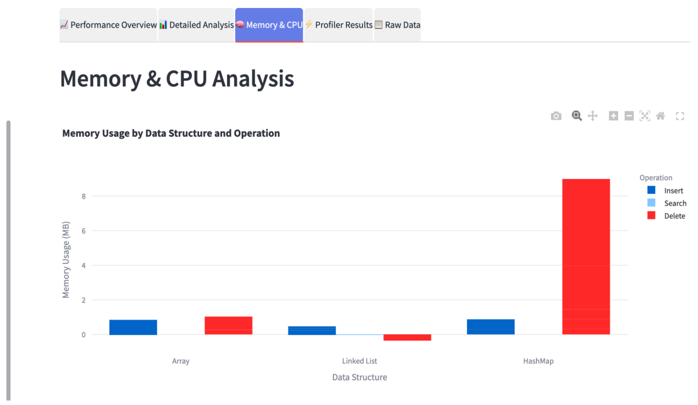
Performance Recommendations

✅ Best Choice: HashMap

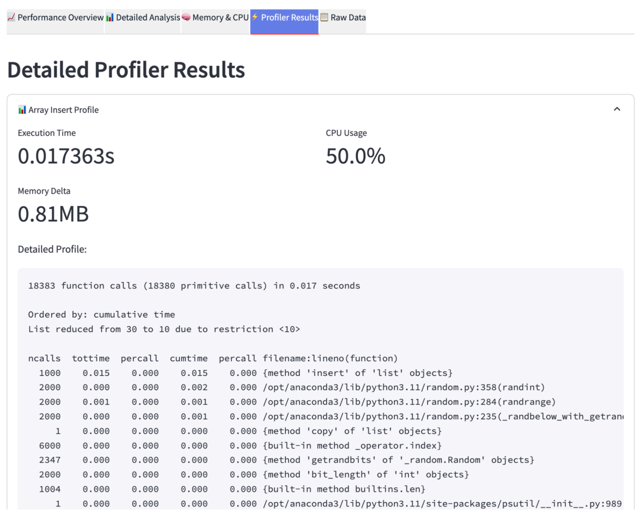
HashMap provides O(1) average time complexity for insertions, making it ideal for scenarios with frequent insertions.

⚠️ Consider: Linked List

Linked List also provides O(1) insertion at the head, but may have higher memory overhead due to pointer storage.



Memory & CPU Analysis



Detailed Profiler Results

📊 Array Insert Profile

Execution Time
0.017363s

CPU Usage
50.0%

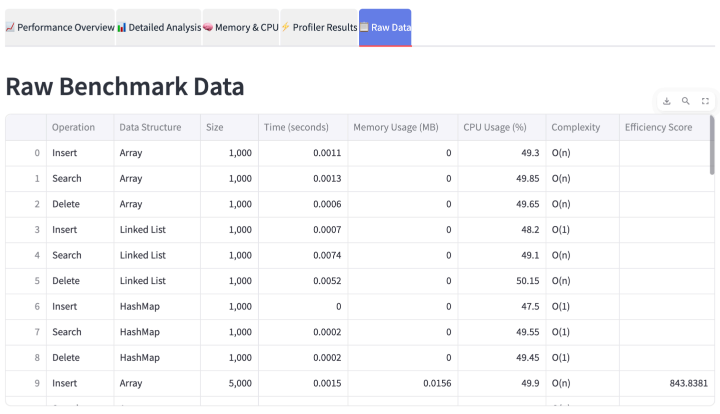
Memory Delta
0.81MB

Detailed Profile:

```
18383 function calls (18380 primitive calls) in 0.017 seconds

Ordered by: cumulative time
List reduced from 30 to 10 due to restriction <10>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  1000    0.015    0.000    0.015    0.000 {method 'insert' of 'list' objects}
  2000    0.000    0.000    0.002    0.000 /opt/anaconda3/lib/python3.11/random.py:358(randint)
  2000    0.001    0.000    0.001    0.000 /opt/anaconda3/lib/python3.11/random.py:284(randrange)
  2000    0.000    0.000    0.001    0.000 /opt/anaconda3/lib/python3.11/random.py:235(_randbelow_with_getran
     1    0.000    0.000    0.000    0.000 {method 'copy' of 'list' objects}
  6000    0.000    0.000    0.000    0.000 {built-in method _operator.index}
  2347    0.000    0.000    0.000    0.000 {method 'getrandbits' of '_random.Random' objects}
  2000    0.000    0.000    0.000    0.000 {method 'bit_length' of 'int' objects}
  1004    0.000    0.000    0.000    0.000 {built-in method builtins.len}
     1    0.000    0.000    0.000    0.000 /opt/anaconda3/lib/python3.11/site-packages/psutil/__init__.py:989
```



Raw Benchmark Data

	Operation	Data Structure	Size	Time (seconds)	Memory Usage (MB)	CPU Usage (%)	Complexity	Efficiency Score
0	Insert	Array	1,000	0.0011	0	49.3	O(n)	
1	Search	Array	1,000	0.0013	0	49.85	O(n)	
2	Delete	Array	1,000	0.0006	0	49.65	O(n)	
3	Insert	Linked List	1,000	0.0007	0	48.2	O(1)	
4	Search	Linked List	1,000	0.0074	0	49.1	O(n)	
5	Delete	Linked List	1,000	0.0052	0	50.15	O(n)	
6	Insert	HashMap	1,000	0	0	47.5	O(1)	
7	Search	HashMap	1,000	0.0002	0	49.55	O(1)	
8	Delete	HashMap	1,000	0.0002	0	49.45	O(1)	
9	Insert	Array	5,000	0.0015	0.0156	49.9	O(n)	843.8381

7. Future Work

1. Add More Data Structures: Expanding the benchmarking tool to include additional data structures such as Binary Search Trees (BSTs), AVL Trees, Heaps (Max and Min), and Graph Representations (Adjacency List, Adjacency Matrix) will provide users with a deeper understanding of how various structures behave under different operations. These data structures are essential for numerous real-world applications like pathfinding, priority queues, and dynamic search trees. Benchmarking these will help in evaluating their practical performance implications, especially in insertion, deletion, traversal, and search tasks.

2. Concurrency Benchmarking: Modern applications often operate in multi-threaded environments, and understanding how data structures behave under concurrent access is crucial for building robust, thread-safe systems. By integrating Python libraries such as `threading`, `concurrent.futures`, and `asyncio`, the benchmarking tool can be extended to simulate real-world concurrent workloads. This will allow users to measure thread safety, race conditions, locking overhead, and the overall scalability of concurrent operations. Profiling under concurrent conditions can uncover bottlenecks and synchronization challenges that single-threaded benchmarks cannot reveal.

3. Real-Time Monitoring: To enhance interactivity and real-time analysis, the addition of live dashboards using Plotly or Bokeh is planned. These dashboards will allow users to visualize metrics such as execution time, CPU usage, and memory consumption in real-time as benchmarks are being executed. Users can dynamically adjust parameters like dataset size, operation type, and data structure without restarting the application. This feature is particularly useful in educational settings and live demonstrations, where immediate feedback is beneficial.

4. Cloud Deployment: Deploying the application on Streamlit Cloud will enable public access, making it easier to share, demo, and collaborate. Cloud deployment ensures platform independence and allows users to run benchmarks on their own machines from a centralized interface. Streamlit's built-in session state management and user authentication can be leveraged to provide personalized experiences. This step also aligns with modern software distribution practices, facilitating continuous updates and wide accessibility.

5. Session Save/Load Functionality: Introducing the ability to save and load benchmark sessions will greatly enhance the tool's utility. Users will be able to store benchmark profiles containing operation details, environment configurations, and results. These profiles can then be loaded to compare results across different time points, machines, or software versions. This feature is essential for longitudinal studies, regression analysis, and collaborative experimentation.

Conclusion of Enhancements: Together, these enhancements will transition the tool from a basic benchmarking script to a powerful, extensible performance analysis suite. It will support both novice learners and experienced developers in making data-driven decisions for system design and optimization.

8. Conclusion

This project successfully bridges theoretical knowledge and practical performance through systematic benchmarking of three core data structures. It uses interactive visualizations, memory and CPU profiling, and raw data tracking to draw meaningful conclusions.

Summary of Achievements:

- Benchmarked performance across insert, search, delete.
- Comparing CPU and memory behavior.
- Used profiler data to locate performance bottlenecks.
- Provided extensible framework with modular design.

Educational and Research Value:

- Highlights real-world implications of data structure selection.
- Useful for teaching algorithm efficiency beyond Big-O notation.
- Ready for enhancements in concurrency, deployment, and scalability.

This tool lays the groundwork for deeper system performance exploration and data structure optimization in both academic and production environments.

9. References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA: MIT Press, 2009.

[2] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Boston, MA: Addison-Wesley, 2011.

[3] Python Software Foundation, "Python Documentation", [Online]. Available: https://docs.python.org/3/. [Accessed: 09-Jul-2025].

[4] Streamlit Inc., "Streamlit Docs", [Online]. Available: https://docs.streamlit.io/. [Accessed: 09-Jul-2025].

[5] Plotly Technologies Inc., "Plotly Python Graphing Library", [Online]. Available: https://plotly.com/python/. [Accessed: 09-Jul-2025].

[6] Psutil Developers, "psutil Python Library", [Online]. Available: https://pypi.org/project/psutil/. [Accessed: 09-Jul-2025].

[7] W. McKinney, *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*, 2nd ed. O'Reilly Media, 2017.

[8] M. Lutz, *Learning Python*, 5th ed. Sebastopol, CA: O'Reilly Media, 2013.

[9] Bokeh Developers, "Bokeh Visualization Library", [Online]. Available: https://docs.bokeh.org/. [Accessed: 09-Jul-2025].

[10] J. D. Hunter, "Matplotlib: A 2D Graphics Environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.