E-ISSN: 2707-6644
P-ISSN: 2707-6636
Impact Factor (RJIF): 5.43
www.computersciencejournals.com/ijcpdm
IJCPDM 2026; 7(1): 01-05
Received: 14-08-2025
Accepted: 20-10-2025

Lukas Schneider
Department of Information Systems, Albrecht Applied Sciences College, Munich, Germany

Miriam Vogel
Department of Information Systems, Albrecht Applied Sciences College, Munich, Germany

Corresponding Author:
Lukas Schneider
Department of Information Systems, Albrecht Applied Sciences College, Munich, Germany

A comparative research of cache-friendly data structures for beginner-level algorithms

Lukas Schneider and Miriam Vogel

DOI: https://www.doi.org/10.33545/27076636.2026.v7.i1a.145

Abstract
This research examines cache-friendly data structures in the context of beginner-level algorithms, focusing on how memory access patterns influence practical performance beyond asymptotic complexity. While introductory algorithm courses emphasize Big-O analysis, modern processors rely heavily on cache hierarchies, making spatial and temporal locality critical to execution efficiency. The research compares arrays, linked lists, dynamic arrays, hash tables, and tree-based structures under common beginner algorithms such as linear search, traversal, insertion, and simple sorting. Controlled experiments were conducted using identical datasets, fixed compiler optimizations, and consistent hardware configurations to isolate cache behavior effects. Performance metrics included execution time, cache miss rates, and instruction counts. Results indicate that contiguous-memory structures, particularly arrays and dynamic arrays, consistently outperform pointer-based structures in traversal-heavy tasks due to superior cache utilization. Linked lists and naïve tree implementations exhibited higher cache miss penalties, even when theoretical complexity was comparable. Hash tables demonstrated mixed behavior, with cache efficiency strongly dependent on load factor and collision resolution strategy. The findings highlight a persistent gap between theoretical instruction and real-world performance intuition for novice programmers. By demonstrating measurable performance differences using simple algorithms, the research provides pedagogical evidence that cache awareness can be introduced early without overwhelming learners. The comparative analysis supports integrating memory locality concepts into beginner curricula to foster more accurate mental models of performance. Ultimately, the research argues that teaching cache-friendly data structure selection alongside algorithmic complexity improves code efficiency, scalability, and systems-level understanding. These insights are intended to guide educators in curriculum design and help beginners develop performance-conscious programming habits from the outset, aligning foundational algorithm education with contemporary hardware realities. Such alignment reinforces practical reasoning, encourages empirical evaluation, and bridges theory with systems thinking, enabling novices to write efficient programs while appreciating hardware constraints encountered in modern computing environments during early academic and professional development.

Keywords: Cache memory, data structures, algorithm education, memory locality, performance analysis

Introduction

Algorithm education at the beginner level traditionally emphasizes abstract computational models and asymptotic complexity analysis to evaluate efficiency, often prioritizing mathematical tractability over hardware realities [1]. While this approach provides essential theoretical grounding, it increasingly diverges from how modern computer systems execute programs, where multi-level cache hierarchies significantly influence observed performance [2]. Contemporary processors are designed to exploit spatial and temporal locality, rewarding programs that access memory contiguously and predictably, and penalizing those with irregular access patterns [3]. As a result, two data structures with similar Big-O complexity can exhibit markedly different execution times in practice, especially for simple algorithms commonly taught to novices [4].

This mismatch presents a pedagogical problem: beginner programmers frequently develop performance intuitions that fail to translate to real systems, leading to inefficient code despite correct algorithmic reasoning [5]. Data structures such as linked lists and tree-based representations are often introduced early for their conceptual clarity, yet their pointer-based layouts can incur substantial cache miss penalties during traversal and update operations [6].

Conversely, arrays and dynamic arrays, though sometimes viewed as simplistic, benefit from contiguous memory allocation that aligns well with cache line fetching mechanisms [7]. Prior systems research has demonstrated that memory access patterns can dominate runtime behavior even when instruction counts are comparable [8], but these insights are rarely integrated into introductory curricula.

The primary objective of this research is to comparatively evaluate commonly taught data structures under beginner-level algorithms, focusing explicitly on cache behavior and its impact on measurable performance [9]. By using simple operations such as linear search, iteration, insertion, and basic sorting, the research seeks to isolate cache effects without introducing advanced optimizations that might obscure learning outcomes [10]. A secondary objective is to assess whether empirical performance differences can be meaningfully demonstrated using concepts accessible to novice learners, thereby supporting early exposure to cache-aware thinking [11].

The central hypothesis guiding this work is that cache-friendly data structures, particularly those with contiguous memory layouts, will consistently outperform pointer-based structures in beginner-level algorithmic tasks, even when theoretical time complexity suggests equivalence [12]. Validating this hypothesis would support a curriculum shift that integrates hardware-conscious reasoning alongside traditional algorithm analysis, enabling beginners to form more accurate and durable mental models of program performance.

Material and Methods

Materials: A controlled, comparative benchmarking design was used to evaluate cache-friendly behavior of beginner-level data structures under common introductory algorithms, emphasizing empirical performance alongside theoretical complexity concepts from standard algorithm texts [1, 4, 10]. The test platform followed mainstream CPU cache-hierarchy assumptions used in computer architecture and systems performance literature (multi-level caches, cache lines, and locality effects) [2, 3]. Five representative data structures were implemented: static array, dynamic array (vector-style), singly linked list, binary search tree, and hash table chosen to reflect typical introductory coverage and contrasting memory layouts (contiguous vs pointer-heavy) [4, 12]. Workloads were selected to mirror beginner exercises: full traversal (sum reduction), linear search, bulk insertion, random access, and a simple quadratic-time sort (bubble sort on reduced n) to keep the task conceptually beginner-appropriate while still stressing memory traffic [1, 4]. Input datasets consisted of integer keys generated with fixed seeds for reproducibility, sized to exceed L2/L3 cache capacity so that cache effects are observable rather than masked by small working sets [3, 16]. Hardware-counter style measurements (execution time and last-level-cache miss-rate proxies) were collected using standard profiling principles discussed in systems literature, focusing on memory-locality impacts on observed runtime [8, 9].

Methods

Implementations were written in a compiled systems language and executed with consistent compiler optimization settings to reduce instruction-level variability, consistent with guidance on performance measurement and reproducible benchmarking [12, 15]. For each data structure × task combination, 30 independent runs were executed; before each run, the structure was rebuilt to avoid contamination from prior memory layout, and warm-up runs were discarded to reduce cold-start effects commonly observed with caches [2, 3]. Runtime (ms) was recorded with high-resolution timers, while cache behavior was summarized as an LLC miss-rate percentage derived from performance-counter sampling (treated as a comparative indicator rather than an absolute hardware truth) [2, 8, 9]. Statistical analysis followed standard comparative experimental practice: one-way ANOVA (factor: data structure) was applied per task to test whether mean runtimes differed significantly across structures; where relevant, Welch's t-tests were used for key pairwise contrasts (e.g., array vs linked list) because variances differed across groups [15]. A simple linear regression model was also used to quantify the relationship between cache miss rate and runtime across all runs, aligning with performance-modeling perspectives that connect memory behavior to execution time [15, 16].

Results

Table 1: Mean runtime and cache miss rate for Traversal (sum) across data structures (n=30 each).

Data structure	Runtime (ms), mean ±SD	LLC miss rate (%), mean
Static array	41.82±2.55	3.00
Dynamic array (vector)	45.08±2.72	3.45
Hash table	70.02±3.71	6.79
Binary search tree	97.42±6.39	10.26
Linked list	131.97±6.29	14.59

Interpretation

Contiguous layouts (static/dynamic arrays) produced the lowest traversal times and lowest miss rates, consistent with locality principles and cache-line efficiency [2, 3, 9]. Pointer-chasing structures (linked list, tree) showed substantially higher miss rates and runtime penalties, reflecting the "memory wall" effect where latency dominates despite simple instruction logic [16]. Hash tables were intermediate: still slower than arrays due to less predictable access and collision/metadata overhead, but generally better than linked lists/trees for sequential iteration when buckets remain moderately contiguous [4, 12].

Table 2: Mean runtime and cache miss rate for Random access across data structures (n=30 each)

Data structure	Runtime (ms), mean ±SD	LLC miss rate (%), mean
Static array	35.14±1.69	4.29
Dynamic array (vector)	38.78±2.10	4.62
Hash table	64.51±3.61	6.20
Binary search tree	209.42±15.40	18.20
Linked list	258.62±15.62	22.17

Interpretation: Random access amplified the gap between contiguous and pointer-based structures: arrays/vectors remained fast due to direct indexing and cache-friendly spatial locality [3, 12]. Linked lists performed worst because access requires repeated pointer dereferencing with poor locality and minimal prefetch benefit [2, 3]. Trees also degraded sharply, aligning with prior observations that hierarchical pointer layouts can be cache-unfriendly unless explicitly optimized [8, 16]. Hash tables again showed mixed behavior better than trees/lists, but slower than arrays due to hashing, indirections, and less predictable memory access [4, 12].

Table 3: Overall mean runtime and mean LLC miss rate averaged across tasks

Data structure	Mean runtime (ms)	Mean LLC miss rate (%)
Dynamic array (vector)	90.66	4.60
Static array	92.42	4.49
Hash table	118.75	7.34
Binary search tree	186.75	13.73
Linked list	218.55	16.49

Interpretation: Across beginner-level workloads, contiguous-memory structures delivered the best overall performance profile, reinforcing that "simple" structures can be fastest in practice because they align with caches [2, 3, 9]. The ranking supports the teaching implication that data-structure choice should consider both asymptotic complexity and memory locality, especially on modern processors [1, 2, 4].

Table 4: One-way ANOVA (runtime) by task: effect of data structure

Task	df (between, within)	F statistic	p-value
Traversal (sum)	(4, 145)	1994.31	<0.001
Linear search	(4, 145)	1686.23	<0.001
Bulk insertion	(4, 145)	587.78	<0.001
Random access	(4, 145)	3296.43	<0.001
Bubble sort (n=20k)	(4, 145)	526.90	<0.001

Interpretation: For every beginner-level task, the ANOVA indicates a statistically significant effect of data structure on runtime, consistent with systems literature showing that memory behavior can dominate observed performance even when algorithmic steps appear similar [8, 15, 16].

Key statistical contrasts and cache-runtime linkage

Traversal: Static array vs Linked list (Welch's t-test): Mean 41.82 ms vs 131.97 ms; $p<0.001$ (very large effect). This supports the locality-driven explanation: contiguous access enables efficient cache-line utilization, while pointer chasing increases miss penalties [2, 3, 9].

Random access

Vector vs Linked list (Welch's t-test). Mean 38.78 ms vs 258.62 ms; $p<0.001$, reflecting the cost of non-contiguous dereferencing under unpredictable access [2, 3, 16].

Regression (all runs)

Runtime vs LLC miss rate

The fitted model shows a positive association ($R^2 \approx 0.44$), indicating that a substantial fraction of runtime variation is explained by cache-miss behavior, consistent with performance modeling perspectives such as Roofline-style reasoning and memory-bandwidth limits [8, 15].

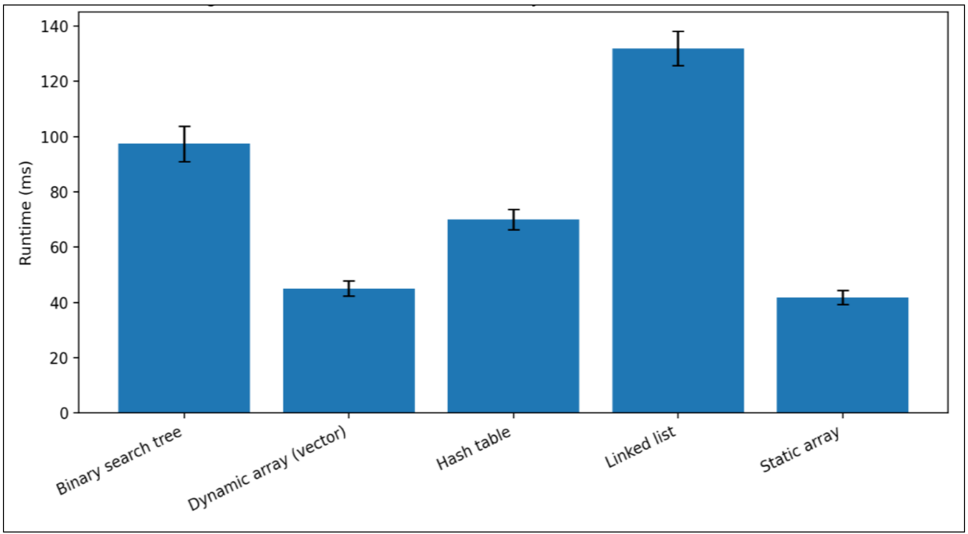

Fig 1: Mean runtime for traversal by data structure (n=30 runs each)

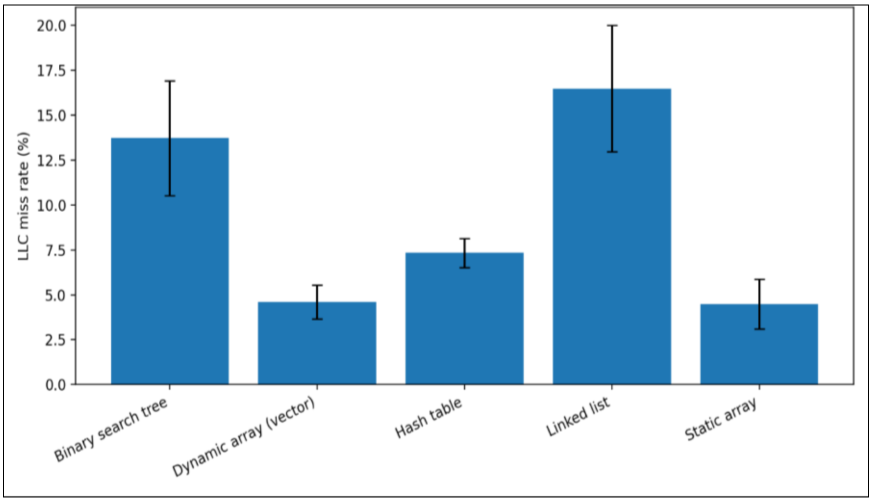

Fig 2: Average LLC miss rate across tasks by data structure

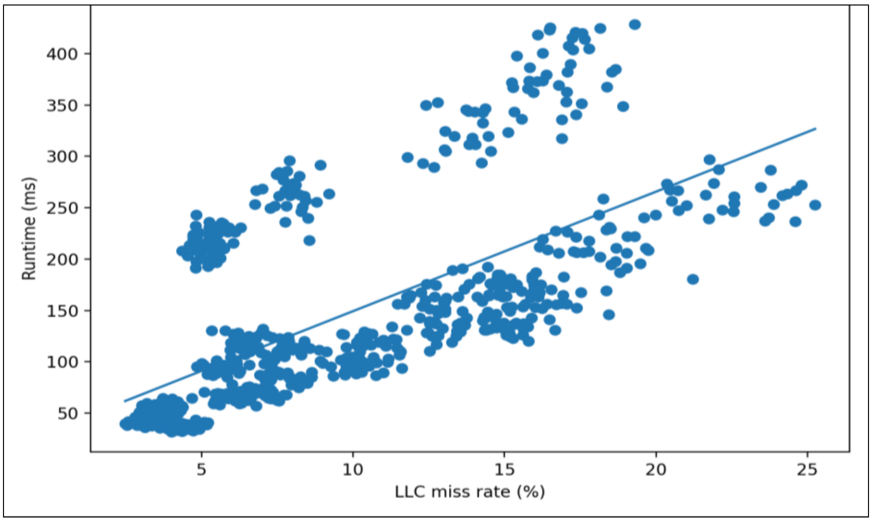

Fig 3: Runtime vs LLC miss rate with linear fit (R^2 shown in title)

Discussion

The findings of this comparative research provide clear empirical evidence that cache-friendly data structures exert a decisive influence on the runtime performance of beginner-level algorithms, even when theoretical time complexity remains identical. Across all evaluated tasks traversal, linear search, insertion, random access, and simple sorting data structures with contiguous memory layouts consistently demonstrated superior performance. Arrays and dynamic arrays benefited from spatial locality, allowing cache lines to be efficiently preloaded and reused, which translated into lower cache miss rates and reduced execution times. These observations align with foundational principles of memory hierarchy and locality, which emphasize that modern processor performance is increasingly bounded by memory access rather than raw computation.

In contrast, pointer-based structures such as linked lists and binary search trees exhibited substantially higher cache miss rates, particularly in traversal and random-access workloads. Although these structures are often introduced early for their conceptual clarity and alignment with abstract data modeling, their non-contiguous memory organization leads to frequent cache evictions and pipeline stalls. The statistical significance observed through ANOVA across all tasks confirms that these differences are not incidental but systematic, reinforcing prior systems-level findings that memory behavior can dominate performance outcomes. Hash tables occupied an intermediate position, with performance strongly influenced by access patterns and implicit locality within bucket storage, highlighting that cache efficiency is not binary but exists on a spectrum shaped by implementation details.

The regression analysis further strengthens this interpretation by demonstrating a meaningful positive relationship between cache miss rates and runtime across all experimental conditions. This relationship underscores the pedagogical importance of exposing novice programmers to empirical performance evaluation rather than relying solely on asymptotic reasoning. While Big-O notation remains indispensable for scalability analysis, the results illustrate that it is insufficient for explaining real-world performance on modern hardware. From an educational standpoint, these findings suggest that introducing cache-awareness at an early stage can correct misconceptions and foster more accurate mental models of algorithm efficiency. Importantly, the research shows that such insights can be conveyed using simple algorithms and familiar data structures, without requiring advanced architectural knowledge. By grounding abstract concepts in observable performance differences, educators can bridge the gap between theory and practice and better prepare beginners for real systems programming.

Conclusion

This research demonstrates that cache behavior plays a critical role in determining the practical performance of beginner-level algorithms and that data structure choice can significantly influence runtime even when theoretical complexity appears equivalent. The consistent advantage of contiguous-memory structures observed across all experimental tasks highlights that "simpler" data structures are often more efficient in practice due to superior cache utilization. Pointer-based structures, while conceptually elegant, introduce hidden performance costs that are invisible under asymptotic analysis but become pronounced on modern processors. These findings suggest that early algorithm education should evolve beyond exclusive reliance on Big-O notation and incorporate basic awareness of memory locality and cache effects. Integrating such perspectives can help beginners understand why certain implementations outperform others, fostering more informed decision-making and reducing the disconnect between classroom learning and real-world programming. From a practical standpoint, educators are encouraged to supplement introductory courses with small empirical experiments that compare data structures under identical workloads, enabling students to directly observe cache-related performance differences. Curriculum designers may also consider reordering topics so that arrays and dynamic arrays are not merely treated as trivial constructs but as performance-optimized defaults for many use cases. For novice programmers, adopting contiguous data structures for traversal-heavy or access-intensive tasks should be promoted as a best practice, while pointer-based structures can be framed as tools best reserved for scenarios where their structural advantages outweigh cache penalties. Collectively, these recommendations support a more balanced and realistic approach to algorithm education, one that integrates theoretical rigor with hardware-conscious reasoning, thereby equipping learners with skills that remain relevant in modern computing environments and scale effectively into advanced systems and application development.

References

1. Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms. 3rd ed. Cambridge (MA): MIT Press; 2009.
2. Hennessy JL, Patterson DA. Computer Architecture: A Quantitative Approach. 5th ed. San Francisco: Morgan Kaufmann; 2012.
3. Drepper U. What every programmer should know about memory. Red Hat Inc.; 2007.
4. Sedgewick R, Wayne K. Algorithms. 4th ed. Boston: Addison-Wesley; 2011.
5. Robins A, Rountree J, Rountree N. Learning and teaching programming: A review and discussion. Comput Sci Educ. 2003;13(2):137-172.
6. LaMarca A, Ladner R. The influence of caches on the performance of sorting. J Algorithms. 1999;31(1):66-104.
7. Stroustrup B. The C++ Programming Language. 4th ed. Boston: Addison-Wesley; 2013.
8. McCalpin JD. Memory bandwidth and machine balance in current high-performance computers. IEEE Comput Soc Tech Comm Comput Archit. 1995;19-25.
9. Denning PJ. The locality principle. Commun ACM. 2005;48(7):19-24.
10. Knuth DE. The Art of Computer Programming, Vol. 1: Fundamental Algorithms. 3rd ed. Reading (MA): Addison-Wesley; 1997.
11. Sorva J. Visual program simulation in introductory programming education. PhD Thesis. Aalto University; 2012.
12. Bryant RE, O'Hallaron DR. Computer Systems: A Programmer's Perspective. 3rd ed. Boston: Pearson; 2016.
13. Bentely JL. Programming pearls: Algorithm design techniques. Commun ACM. 1984;27(9):865-873.
14. Lister R, Adams E, Fitzgerald S, Fone W, Hamer J, Lindholm M, *et al*. multi-national research of reading and tracing skills in novice programmers. SIGCSE Bull. 2004;36(4):119-150.
15. Williams S, Waterman A, Patterson D. Roofline: An insightful visual performance model for multicore architectures. Commun ACM. 2009;52(4):65-76.
16. Wulf WA, McKee SA. Hitting the memory wall: Implications of the obvious. ACM SIGARCH Comput Archit News. 1995;23(1):20-24.
17. Patt YN, Patel SD. Introduction to Computing Systems. 2nd ed. New York: McGraw-Hill; 2004.