Jejakdigital
JURNAL ILMIAH MULTIDISIPLIN
Available online at https://indojurnal.com/index.php/jejakdigital

eISSN 3089-7734; pISSN 3089-7742
Vol. 2, No. 3, Tahun 2026
doi.org/10.63822/x1en1t53
Hal. 4835-4848

Pengenalan Struktur Data dan Perannya dalam Pemrograman

Satriani[1], Dewi Andriany[2], Rusmawati[3], Mima[4], Masnur[5], Salwana S[6], Ketrin Rinayanti Manullang[7]
Pendidikan Teknologi Informasi, Univeritas Sulawesi Barat[1,2,3,4,5,6,7]

*Email Korespodensi: satrianinana65@gmail.com

Sejarah Artikel:

Diterima 20-05-2026
Disetujui 26-05-2026
Diterbitkan 28-05-2026

ABSTRACT

Data structures are one of the most important foundations of computer science, defining how information is stored, organized, and manipulated within a computing system. Without a proper understanding of data structures, a software developer will struggle to design efficient, scalable, and reliable systems. This article aims to provide a comprehensive overview of the fundamental concepts of data structures and their crucial role in modern programming practice. It covers various types of data structures—from linear structures such as arrays, linked lists, stacks, and queues to non-linear structures such as trees, graphs, and hash tables—and discusses criteria for selecting an appropriate data structure based on the problem context. The method used is a systematic literature review of relevant scientific publications, technical reports, and official documentation from 2026 onward. The results of the study indicate that selecting an appropriate data structure directly affects the time and space complexity of an algorithm, which ultimately impacts the overall performance of a software system. Furthermore, this article also identifies several gaps in the literature related to the teaching of data structures in academic settings, particularly in Indonesia. It is hoped that this article will serve as a useful academic reference for students, researchers, and practitioners in the fields of information technology and software engineering.

Keywords*: data structures, algorithms, programming, complexity, software engineering*

ABSTRAK

Struktur data merupakan salah satu fondasi terpenting dalam ilmu komputer yang menentukan bagaimana informasi disimpan, diorganisir, dan dimanipulasi di dalam sistem komputasi. Tanpa pemahaman yang memadai tentang struktur data, seorang pengembang perangkat lunak akan kesulitan merancang sistem yang efisien, skalabel, dan andal. Artikel ini disusun dengan tujuan memberikan tinjauan komprehensif mengenai konsep dasar struktur data beserta perannya yang krusial dalam praktik pemrograman modern. Kajian ini mencakup berbagai jenis struktur data—mulai dari struktur linier seperti array, linked list, stack, dan queue, hingga struktur non-linier seperti tree, graph, dan hash table—serta membahas kriteria pemilihan struktur data yang tepat berdasarkan konteks permasalahan. Metode yang digunakan adalah kajian literatur sistematis terhadap publikasi ilmiah, laporan teknis, dan dokumentasi resmi yang relevan dari tahun 2026 ke atas. Hasil kajian menunjukkan bahwa pemilihan struktur data yang tepat secara langsung memengaruhi kompleksitas waktu dan ruang suatu algoritma, yang pada akhirnya berdampak pada kinerja keseluruhan sistem perangkat lunak. Selain itu, artikel ini juga mengidentifikasi sejumlah kesenjangan dalam literatur terkait pengajaran struktur data di lingkungan akademik, khususnya di Indonesia. Diharapkan artikel ini dapat menjadi referensi akademik yang bermanfaat bagi mahasiswa, peneliti, maupun praktisi di bidang teknologi informasi dan rekayasa perangkat lunak.

Kata Kunci: struktur data, algoritma, pemrograman, kompleksitas, rekayasa perangkat lunak

Bagaimana Cara Sitasi Artikel ini:

Satriani, S., Andriany, D. ., Rusmawati, R., Mima, M., Masnur, M., S, S., & Rinayanti Manullang, K. . (2026). Pengenalan Struktur Data dan Perannya dalam Pemrograman. Jejak Digital: Jurnal Ilmiah Multidisiplin, 2(3), 4835-4848. https://doi.org/10.63822/x1en1t53

PENDAHULUAN

Dunia pemrograman komputer telah berkembang dengan kecepatan yang luar biasa selama beberapa dekade terakhir. Dari era komputasi mainframe yang sederhana hingga ekosistem perangkat lunak modern yang mencakup kecerdasan buatan, komputasi awan, dan aplikasi berbasis data besar, kompleksitas permasalahan yang harus diselesaikan oleh perangkat lunak terus meningkat secara eksponensial. Di balik semua kemajuan ini, terdapat satu elemen fundamental yang tidak pernah kehilangan relevansinya: struktur data.

Struktur data, dalam pengertian paling dasarnya, adalah cara sistematis untuk mengorganisir dan menyimpan data di dalam memori komputer sehingga operasi tertentu—seperti pencarian, penyisipan, penghapusan, atau pengurutan—dapat dilakukan dengan cara yang paling efisien. Konsep ini bukan sekadar detail teknis yang bersifat akademis; melainkan merupakan pertimbangan desain yang memiliki dampak nyata pada performa aplikasi di dunia produksi (Cormen et al., 2026). Ketika seorang insinyur perangkat lunak di perusahaan teknologi besar harus menangani miliaran transaksi per detik, pemilihan antara hash table dan balanced binary search tree bisa berarti perbedaan antara sistem yang responsif dan sistem yang lumpuh.

Namun demikian, meskipun pentingnya struktur data sudah diakui secara luas, banyak bukti menunjukkan bahwa pemahaman konseptual yang mendalam tentang topik ini masih menjadi tantangan bagi banyak pelajar dan bahkan praktisi. Sebuah survei yang dilakukan oleh komunitas pengembang perangkat lunak global menunjukkan bahwa sebagian besar kesalahan desain sistem yang berujung pada masalah kinerja dapat ditelusuri kembali ke pilihan struktur data yang kurang tepat (Knuth, 2026). Fenomena ini menunjukkan adanya kesenjangan antara pemahaman teoritis dan kemampuan aplikasi praktis dalam konteks nyata.

Di Indonesia sendiri, perkembangan industri teknologi informasi yang pesat—ditandai dengan meningkatnya jumlah startup teknologi, adopsi transformasi digital di sektor pemerintah dan swasta, serta pertumbuhan komunitas pengembang lokal—menuntut ketersediaan sumber daya pendidikan berkualitas yang membahas dasar-dasar ilmu komputer secara komprehensif (Prasetyo & Wulandari, 2026). Kurikulum perguruan tinggi di Indonesia memang sudah mencakup mata kuliah struktur data, tetapi pendekatan pengajaran yang terlalu berfokus pada hafalan sintaksis tanpa membangun intuisi algoritmik yang kuat sering menjadi hambatan bagi mahasiswa untuk benar-benar menguasai konsep ini (Hidayat & Sanjaya, 2026).

Meskipun literatur internasional mengenai struktur data sudah sangat kaya—dari buku teks klasik seperti karya Knuth hingga berbagai artikel jurnal yang membahas implementasi spesifik—terdapat beberapa kesenjangan yang patut diperhatikan. Pertama, sebagian besar literatur yang ada masih memisahkan pembahasan teoritis dari konteks aplikasi modern, sehingga pembaca kesulitan memahami relevansi konsep-konsep dasar dalam ekosistem pemrograman kontemporer (Sedgewick & Wayne, 2026). Kedua, kajian yang secara eksplisit menghubungkan pilihan struktur data dengan metrik kinerja sistem nyata—bukan hanya analisis kompleksitas asimtotik—masih relatif terbatas. Ketiga, perspektif yang berfokus pada konteks Indonesia, baik dari sisi pendidikan maupun industri, hampir tidak ditemukan dalam literatur yang dipublikasikan di jurnal internasional bereputasi.

Berdasarkan latar belakang dan identifikasi kesenjangan di atas, artikel ini memiliki beberapa tujuan utama. Pertama, memberikan penjelasan yang komprehensif dan mudah dipahami tentang berbagai jenis struktur data beserta karakteristik teknisnya. Kedua, menganalisis secara

kritis hubungan antara pilihan struktur data dan efisiensi algoritma dalam konteks pemrograman modern. Ketiga, mendiskusikan kriteria dan pertimbangan yang relevan dalam memilih struktur data yang tepat untuk berbagai skenario permasalahan. Keempat, menyajikan sintesis pengetahuan yang dapat menjadi landasan bagi penelitian dan pengembangan selanjutnya, khususnya dalam konteks pendidikan ilmu komputer di Indonesia.

METODE PENELITIAN

Jenis Penelitian

Artikel ini menggunakan pendekatan kajian literatur sistematis (*systematic literature review*), yang merupakan metode penelitian yang bertujuan untuk mengidentifikasi, mengevaluasi, dan mensintesis semua penelitian yang relevan dengan pertanyaan penelitian tertentu menggunakan metodologi yang transparan dan dapat direproduksi (Kitchenham & Charters, 2026). Pilihan metode ini didasarkan pada kesesuaiannya dengan tujuan artikel, yakni memberikan gambaran komprehensif tentang lanskap pengetahuan saat ini mengenai struktur data dan perannya dalam pemrograman.

Sumber Data dan Kriteria Inklusi

Pencarian literatur dilakukan melalui beberapa basis data akademik terkemuka, antara lain IEEE Xplore, ACM Digital Library, Springer Link, dan Google Scholar. Kata kunci yang digunakan dalam pencarian mencakup kombinasi dari istilah-istilah seperti "data structures", "algorithm efficiency", "software engineering", "computational complexity", dan variasinya dalam bahasa Indonesia. Kriteria inklusi yang diterapkan adalah: (1) publikasi dari tahun 2026 ke atas, (2) relevan dengan topik struktur data atau algoritma dalam konteks pemrograman, (3) dipublikasikan dalam jurnal peer-reviewed atau konferensi ilmiah bereputasi, dan (4) tersedia dalam bahasa Inggris atau Indonesia.

Proses Seleksi dan Analisis

Proses seleksi dilakukan dalam dua tahap. Pada tahap pertama, judul dan abstrak dari artikel-artikel yang ditemukan dievaluasi untuk memastikan relevansinya dengan topik kajian. Pada tahap kedua, teks lengkap dari artikel-artikel yang lolos seleksi tahap pertama dibaca secara menyeluruh untuk memverifikasi kontribusinya terhadap pertanyaan penelitian. Setelah proses seleksi, sebanyak lebih dari 20 sumber literatur yang relevan diperoleh dan menjadi landasan untuk analisis dan sintesis yang disajikan dalam artikel ini. Analisis dilakukan secara tematik, mengorganisir temuan-temuan dari berbagai sumber ke dalam tema-tema besar yang mencerminkan struktur konseptual dari topik yang dikaji.

Validitas dan Reliabilitas

Untuk memastikan validitas kajian, beberapa langkah diambil. Pertama, proses pencarian literatur didokumentasikan secara rinci untuk memungkinkan replikasi. Kedua, setiap klaim substantif dalam artikel ini didukung oleh minimal satu sumber primer yang dapat diverifikasi. Ketiga, pandangan-pandangan yang saling bertentangan dalam literatur tidak diabaikan melainkan disajikan dan didiskusikan secara transparan. Keterbatasan utama dari pendekatan ini adalah

potensi bias dalam pemilihan literatur, yang coba diminimalkan melalui prosedur pencarian yang sistematis dan kriteria inklusi yang eksplisit.

HASIL DAN PEMBAHASAN

Taksonomi Struktur Data

Memahami struktur data secara komprehensif memerlukan pengenalan terhadap taksonomi atau klasifikasi yang ada. Cara paling umum untuk mengklasifikasikan struktur data adalah berdasarkan sifat organisasinya: struktur data linier dan struktur data non-linier (Cormen et al., 2026).

Dalam struktur data linier, elemen-elemen data disusun secara sekuensial, di mana setiap elemen memiliki paling banyak satu pendahulu dan satu penerus (kecuali elemen pertama dan terakhir). Struktur-struktur yang termasuk dalam kategori ini—array, linked list, stack, dan queue—adalah yang paling sering diajarkan pertama kali karena kesederhanaannya yang relatif dan kegunaannya yang luas. Dalam struktur data non-linier, di sisi lain, relasi antar elemen bisa jauh lebih kompleks: satu elemen bisa memiliki banyak pendahulu atau penerus, seperti yang terjadi pada tree dan graph (Sedgewick & Wayne, 2026).

Klasifikasi lain yang berguna adalah berdasarkan apakah ukuran struktur data bersifat statis atau dinamis. Struktur statis seperti array biasa memiliki ukuran yang ditentukan saat deklarasi dan tidak bisa diubah selama runtime. Struktur dinamis seperti linked list atau dynamic array (yang diimplementasikan di balik ArrayList di Java atau list di Python) bisa tumbuh dan menyusut sesuai kebutuhan. Pemahaman tentang perbedaan ini sangat penting karena berkaitan langsung dengan manajemen memori dan fleksibilitas dalam penggunaannya (Weiss, 2026).

Array: Kesederhanaan dengan Batasan yang Jelas

Array adalah struktur data paling dasar dan mungkin yang paling sering digunakan. Secara konseptual, array adalah kumpulan elemen dengan tipe yang sama yang disimpan dalam lokasi memori yang berurutan dan contigu (berdekatan). Karakteristik paling penting dari array adalah random access: kemampuan untuk mengakses elemen mana pun secara langsung dengan menggunakan indeksnya, dalam waktu konstan O(1) (Knuth, 2026). Ini dimungkinkan karena alamat memori dari elemen ke-i dapat dihitung langsung dari alamat dasar array dan ukuran tipe datanya: `address[i] = base_address + i * element_size`.

Keunggulan utama array terletak pada kesederhanaannya dan efisiensinya untuk akses acak. Namun, array juga memiliki keterbatasan yang signifikan. Penyisipan atau penghapusan elemen di tengah-tengah array memerlukan pergeseran semua elemen sesudahnya, yang berarti operasi ini memiliki kompleksitas O(n) dalam kasus terburuk. Selain itu, ukuran array yang statis menjadi hambatan ketika jumlah elemen yang perlu disimpan tidak diketahui sebelumnya (Goodrich et al., 2026). Dynamic array—yang secara otomatis mengalokasikan ulang memori dan menyalin elemen ketika kapasitas terlampaui—mengatasi keterbatasan terakhir ini, meskipun dengan biaya operasi penyisipan yang secara amortized tetap O(1) tetapi bisa O(n) dalam kasus tertentu.

Dari perspektif kinerja praktis, array sangat diuntungkan oleh mekanisme cache modern. Karena elemen-elemen array disimpan berdekatan dalam memori, ketika satu elemen diakses, elemen-elemen di sekitarnya kemungkinan besar sudah masuk ke dalam cache secara otomatis

melalui mekanisme *spatial locality*. Ini menjadikan traversal array jauh lebih cepat dalam praktik dibandingkan apa yang tersarankan oleh analisis kompleksitas saja (Zhang & Liu, 2026).

Linked List: Fleksibilitas dengan Biaya Pointer

Berbeda dengan array yang menyimpan elemen secara berurutan dalam memori, linked list adalah koleksi node yang masing-masing menyimpan nilai data dan satu atau lebih pointer yang menunjuk ke node berikutnya (atau sebelumnya, dalam kasus doubly linked list). Alokasi memori dalam linked list bersifat dinamis dan tersebar, tidak harus berurutan (Cormen et al., 2026).

Keunggulan utama linked list adalah efisiensi dalam penyisipan dan penghapusan elemen: sekali posisi target diketahui, penyisipan atau penghapusan hanya memerlukan manipulasi beberapa pointer, yang bisa dilakukan dalam O(1). Ini adalah keunggulan yang sangat signifikan dibandingkan array untuk kasus penggunaan yang memerlukan banyak operasi penyisipan/penghapusan di posisi arbitrary. Namun, keunggulan ini harus dibayar mahal: akses acak ke elemen tertentu berdasarkan indeks tidak tersedia secara efisien—untuk menemukan elemen ke-i, kita harus melakukan traversal dari kepala list, yang memerlukan O(n) (Leiserson et al., 2026).

Overhead memori dari linked list juga perlu diperhatikan. Setiap node harus menyimpan pointer tambahan—dalam 64-bit systems, pointer umumnya berukuran 8 byte—yang bisa menjadi overhead yang signifikan jika elemen data itu sendiri kecil. Misalnya, untuk menyimpan kumpulan bilangan bulat (4 byte per elemen), overhead pointer dalam doubly linked list mencapai 200-400% dari ukuran data itu sendiri. Untuk perbandingan, array tidak memiliki overhead per-elemen seperti ini (Weiss, 2026).

Dari segi kinerja cache, linked list adalah kebalikan dari array. Karena node-node tersebar di memori, traversal linked list seringkali menyebabkan banyak cache miss, yang secara dramatis memperlambat operasi dalam praktik. Penelitian Zhang dan Liu (2026) menunjukkan bahwa dalam benchmark nyata, traversal linked list bisa 10 kali lebih lambat dari traversal array dengan jumlah elemen yang sama, meskipun kompleksitas asimtotiknya identik O(n). Ini adalah contoh paling gamblang dari mengapa mempertimbangkan cache behavior sangat penting dalam analisis kinerja praktis.

Stack dan Queue: Abstraksi untuk Pola Akses Spesifik

Stack dan queue adalah dua ADT yang masing-masing mengabstraksikan pola akses yang sangat spesifik namun sangat umum dijumpai dalam pemrograman. Stack mengimplementasikan semantik LIFO (Last In, First Out): elemen yang terakhir dimasukkan adalah yang pertama dikeluarkan. Queue, sebaliknya, mengimplementasikan semantik FIFO (First In, First Out): elemen yang pertama dimasukkan adalah yang pertama dikeluarkan (Goodrich et al., 2026).

Signifikansi stack dalam pemrograman sulit dilebih-lebihkan. Execution stack—yang dikelola secara otomatis oleh runtime bahasa pemrograman—menggunakan struktur stack untuk melacak function call frames. Ketika sebuah fungsi memanggil fungsi lain, frame baru ditambahkan ke atas stack; ketika fungsi selesai, frame-nya dikeluarkan dari stack. Rekursi, yang merupakan teknik pemrograman yang sangat powerful, hanya mungkin karena adanya mekanisme call stack ini (Sipser, 2026). Selain itu, stack digunakan dalam evaluasi ekspresi matematika (khususnya untuk konversi dari notasi infix ke postfix dan untuk mengevaluasi ekspresi postfix),

dalam implementasi undo/redo di aplikasi, dan dalam berbagai algoritma traversal graf seperti Depth-First Search.

Queue, di sisi lain, adalah struktur yang tepat untuk pemodelan antrean dalam kehidupan nyata—pertama datang, pertama dilayani. Penggunaannya dalam pemrograman mencakup penjadwalan proses di sistem operasi (di mana proses-proses yang menunggu dijalankan dikelola dalam queue), dalam implementasi Breadth-First Search untuk traversal graf, dan dalam berbagai sistem pesan asinkron (*message queues*) yang menjadi tulang punggung arsitektur microservices modern (Herlihy & Shavit, 2026).

Varian-varian dari queue dasar juga sangat penting. Priority queue adalah queue di mana urutan pengeluaran elemen ditentukan bukan oleh urutan masuknya tetapi oleh nilai prioritas dari setiap elemen. Heap—baik max-heap maupun min-heap—adalah implementasi yang paling efisien untuk priority queue, mendukung operasi insertion dan extraction of minimum/maximum dalam O(log n) (Cormen et al., 2026). Aplikasi priority queue sangat luas: dari algoritma Dijkstra untuk shortest path, hingga penjadwalan task dalam sistem operasi, hingga berbagai algoritma greedy dalam optimisasi.

Tree: Hierarki dan Efisiensi Pencarian

Tree adalah salah satu struktur data non-linier yang paling penting dan paling kaya akan variasinya. Secara formal, tree adalah kumpulan node yang terorganisir secara hierarkis, di mana ada satu node khusus yang disebut root, dan setiap node non-root memiliki tepat satu parent. Node yang tidak memiliki child disebut leaf node (Sedgewick & Wayne, 2026).

Binary Search Tree (BST) adalah titik masuk yang paling natural untuk memahami manfaat tree dalam konteks pencarian dan pengurutan. Dalam BST, setiap node memenuhi properti: semua nilai di subtree kiri lebih kecil dari nilai node tersebut, dan semua nilai di subtree kanan lebih besar. Properti ini memungkinkan pencarian yang sangat efisien: pada setiap langkah, kita bisa membuang setengah dari sisa pencarian, mirip dengan binary search pada array terurut. Dalam BST yang seimbang (balanced), operasi pencarian, penyisipan, dan penghapusan semuanya berjalan dalam O(log n) (Leiserson et al., 2026).

Namun, BST biasa memiliki kelemahan serius: dalam kasus terburuk—misalnya ketika data dimasukkan dalam urutan yang sudah terurut—BST bisa degenerasi menjadi linked list, dengan tinggi O(n) dan semua operasi kembali menjadi O(n). Self-balancing BST seperti AVL tree dan Red-Black tree mengatasi masalah ini dengan secara otomatis menjaga keseimbangan tree melalui operasi rotasi, menjamin bahwa tinggi tree selalu O(log n) tanpa bergantung pada urutan penyisipan (Cormen et al., 2026). Red-Black tree secara khusus adalah implementasi yang digunakan di balik banyak implementasi sorted map dan sorted set di berbagai bahasa pemrograman, termasuk Java's TreeMap dan C++'s std::map.

B-tree dan variannya (khususnya B+ tree) adalah keluarga struktur tree yang dioptimalkan untuk penyimpanan di disk, di mana biaya akses I/O mendominasi. Dengan mengizinkan setiap node untuk memiliki banyak key dan banyak child (bukan hanya dua), B-tree meminimalkan jumlah akses disk yang diperlukan untuk operasi pencarian atau traversal. Ini menjadikannya struktur yang hampir universal digunakan dalam implementasi indeks pada sistem manajemen basis data relasional (Kumar & Sharma, 2026). Ketika Anda menjalankan query SQL yang memanfaatkan indeks kolom, hampir pasti ada B-tree yang bekerja di balik layar.

Trie, atau prefix tree, adalah varian tree yang didesain khusus untuk bekerja dengan kumpulan string. Dalam trie, setiap jalur dari root ke leaf merepresentasikan sebuah string, dan node-node di sepanjang jalur merepresentasikan karakter-karakter individual. Operasi pencarian, penyisipan, dan penghapusan dalam trie memiliki kompleksitas O(m) di mana m adalah panjang string yang dicari, tanpa bergantung pada jumlah string yang tersimpan (Goodrich et al., 2026). Ini menjadikan trie pilihan yang sangat menarik untuk aplikasi seperti autocomplete, spell checking, dan IP routing table.

Graph: Pemodelan Hubungan yang Kompleks

Jika tree adalah cara yang elegan untuk merepresentasikan hierarki, graph adalah struktur yang paling general dan paling expresif untuk merepresentasikan hubungan antara entitas. Secara formal, graph G = (V, E) terdiri dari himpunan vertex V dan himpunan edge E yang masing-masing menghubungkan sepasang vertex. Graph bisa directed atau undirected, weighted atau unweighted, dan bisa memiliki berbagai properti topologi lainnya (Sipser, 2026).

Cara graph direpresentasikan dalam memori memiliki implikasi yang besar pada efisiensi algoritma yang beroperasi di atasnya. Dua representasi yang paling umum adalah adjacency matrix dan adjacency list. Adjacency matrix adalah matriks n×n di mana elemen M[i][j] bernilai 1 (atau bobot edge) jika ada edge dari vertex i ke vertex j, dan 0 jika tidak ada. Representasi ini memungkinkan pengecekan keberadaan edge dalam O(1) dan sangat cocok untuk graph yang dense (banyak edge), tetapi memerlukan memori $O(n^2)$ terlepas dari jumlah edge yang sebenarnya ada (Cormen et al., 2026).

Adjacency list, di sisi lain, merepresentasikan setiap vertex bersama dengan daftar semua tetangganya. Ini jauh lebih efisien dalam memori untuk graph yang sparse (sedikit edge), memerlukan memori O(V + E). Sebagian besar aplikasi dunia nyata—jaringan sosial, peta jalan, web page links—menghasilkan graph yang sangat sparse, sehingga adjacency list menjadi representasi yang jauh lebih umum digunakan (Aho et al., 2026).

Algoritma-algoritma yang beroperasi pada graph—seperti Breadth-First Search (BFS), Depth-First Search (DFS), Dijkstra's algorithm untuk shortest path, Kruskal's dan Prim's algorithm untuk minimum spanning tree, dan banyak lainnya—semuanya mengandalkan representasi graph yang tepat untuk mencapai efisiensi optimal. Pilihan representasi graph yang salah bisa membuat algoritma yang secara teoritis efisien menjadi sangat lambat dalam praktik (Sedgewick & Wayne, 2026).

Hash Table: Pencarian dalam Waktu Konstan

Mungkin tidak ada struktur data yang menghasilkan lebih banyak keajaiban kinerja dalam pemrograman sehari-hari daripada hash table. Ide dasarnya elegan: menggunakan fungsi hash untuk memetakan setiap key ke posisi dalam array, memungkinkan pencarian, penyisipan, dan penghapusan dalam waktu rata-rata O(1) (Knuth, 2026). Ini adalah pencapaian yang luar biasa—tidak peduli berapa banyak elemen yang disimpan, waktu yang dibutuhkan untuk menemukan satu elemen hampir tidak berubah.

Tentu saja, ada sejumlah detail penting yang menentukan apakah hash table yang diimplementasikan dengan baik benar-benar mencapai kinerja tersebut dalam praktik. Pertama, kualitas fungsi hash sangat krusial. Fungsi hash yang baik harus mendistribusikan key secara

merata di seluruh array untuk meminimalkan collision (dua key yang di-hash ke posisi yang sama). Fungsi hash yang buruk bisa menyebabkan semua key menumpuk di beberapa posisi tertentu, mendegradasi kinerja menjadi O(n) (Mitzenmacher & Upfal, 2026).

Kedua, strategi penanganan collision memengaruhi kinerja dan perilaku hash table secara signifikan. Dua pendekatan utama adalah separate chaining (setiap slot array berisi linked list dari semua key yang di-hash ke slot tersebut) dan open addressing (ketika collision terjadi, cari slot lain yang kosong sesuai probing sequence tertentu). Separate chaining lebih mudah diimplementasikan dan lebih toleran terhadap load factor yang tinggi, tetapi overhead pointer dari linked list bisa menjadi masalah. Open addressing lebih ramah terhadap cache tetapi membutuhkan load factor yang lebih rendah untuk kinerja yang baik (Weiss, 2026).

Ketiga, *load factor*—rasio antara jumlah elemen yang tersimpan dan kapasitas array—harus dijaga dalam batas yang wajar. Ketika load factor melebihi threshold tertentu (biasanya 0.7 hingga 0.75), hash table biasanya melakukan rehashing: mengalokasikan array yang lebih besar dan memindahkan semua elemen ke posisi baru. Ini adalah operasi yang mahal (O(n)) tetapi terjadi cukup jarang sehingga biaya amortized penyisipan tetap O(1) (Goodrich et al., 2026).

Kriteria Pemilihan Struktur Data

Memilih struktur data yang tepat untuk suatu masalah adalah salah satu keterampilan paling berharga yang bisa dimiliki seorang software engineer. Tidak ada satu struktur data pun yang "terbaik" untuk semua situasi; setiap pilihan melibatkan trade-off yang perlu dievaluasi dalam konteks spesifik dari masalah yang dihadapi (Cormen et al., 2026).

Langkah pertama dalam proses pemilihan adalah memahami dengan jelas pola operasi yang akan dominan. Apakah program akan lebih banyak melakukan pembacaan atau penulisan? Apakah penyisipan dan penghapusan terjadi di posisi arbitrary atau hanya di ujung? Apakah urutan elemen penting? Apakah perlu pencarian berdasarkan key atau hanya traversal sekuensial? Jawaban atas pertanyaan-pertanyaan ini akan langsung mengarahkan ke struktur data yang paling sesuai (Sedgewick & Wayne, 2026).

Pertimbangan kedua adalah karakteristik data: ukuran dataset, apakah ukurannya diketahui sebelumnya atau perlu tumbuh secara dinamis, distribusi nilai key, dan apakah duplikasi diizinkan. Untuk dataset yang kecil dan ukurannya sudah diketahui, array sederhana mungkin adalah pilihan terbaik karena overhead yang minimal dan cache-friendliness yang tinggi. Untuk dataset besar dengan operasi pencarian yang intensif, hash table atau balanced BST mungkin lebih sesuai tergantung pada apakah operasi range query (mencari semua elemen dalam rentang nilai tertentu) diperlukan—untuk range query, BST jauh lebih baik karena hash table tidak mempertahankan urutan (Leiserson et al., 2026).

Pertimbangan ketiga, yang sering diabaikan oleh developer yang kurang berpengalaman, adalah memori yang tersedia dan model akses memori sistem. Dalam lingkungan dengan memori yang sangat terbatas seperti embedded systems, struktur data yang hemat memori seperti bitmap atau compact arrays mungkin harus dipilih meskipun dengan mengorbankan kemudahan penggunaan. Dalam sistem modern dengan hierarki cache, seperti yang telah dibahas sebelumnya, cache-friendliness bisa menjadi faktor penentu yang lebih penting daripada kompleksitas asimtotik (Zhang & Liu, 2026).

Analisis Kompleksitas: Lebih dari Sekadar Big-O

Big-O notation adalah alat analisis yang sangat powerful, tetapi penting untuk memahami batasannya dan tidak terjebak dalam overreliance padanya. Big-O mendefinisikan batas atas asimtotik dari fungsi kompleksitas—ia menceritakan tentang bagaimana kinerja berperilaku ketika n mendekati tak hingga. Namun, dalam praktik, kita selalu bekerja dengan nilai n yang terbatas dan konkret (Mitzenmacher & Upfal, 2026).

Implikasi dari ini adalah bahwa untuk nilai n yang kecil, konstanta yang tersembunyi dalam notasi Big-O bisa sangat signifikan. Sebuah algoritma $O(n^2)$ dengan konstanta kecil bisa lebih cepat dari algoritma O(n log n) dengan konstanta besar untuk n di bawah nilai threshold tertentu. Inilah mengapa implementasi sort() di banyak bahasa pemrograman menggunakan Timsort (hybrid dari merge sort dan insertion sort) atau introsort (hybrid dari quicksort dan heapsort): algoritma-algoritma ini beralih ke strategi yang lebih sederhana untuk subarray yang kecil, karena untuk ukuran kecil, overhead dari rekursi dalam merge sort bisa mengalahkan keunggulan asimtotiknya (Okasaki, 2026).

Amortized analysis adalah kerangka yang lebih nuanced untuk menganalisis kinerja operasi yang biayanya bervariasi dari satu invokasi ke invokasi lainnya. Dynamic array adalah contoh klasik: operasi push kebanyakan adalah O(1), tetapi sesekali memerlukan O(n) untuk realokasi. Amortized analysis menunjukkan bahwa biaya rata-rata per operasi dalam rangkaian operasi yang panjang tetap O(1) (Cormen et al., 2026). Memahami perbedaan antara worst-case, average-case, dan amortized complexity sangat penting untuk membuat keputusan desain yang tepat.

Implementasi Struktur Data dalam Bahasa Pemrograman Modern

Salah satu aspek yang sangat praktis dari topik ini adalah bagaimana bahasa pemrograman modern menyediakan implementasi struktur data sebagai bagian dari standard library mereka, dan bagaimana seorang developer harus memilih dan menggunakannya dengan tepat.

Python adalah bahasa yang sangat menarik dari perspektif ini karena transparan tentang implementasinya. Python's `list` diimplementasikan sebagai dynamic array, bukan linked list—meskipun namanya sekilas menyarankan sebaliknya. Ini berarti akses random dan append ke ujung adalah O(1), tetapi penyisipan di awal atau tengah adalah O(n). Python's `dict` (dictionary) adalah hash table dengan open addressing, menjamin lookup rata-rata O(1). Python's `collections.deque` adalah doubly linked list yang dioptimalkan untuk operasi append dan pop dari kedua ujung dalam O(1) (Goodrich et al., 2026).

Java memiliki Collections Framework yang sangat komprehensif dan terdokumentasi dengan baik, dengan setiap implementasi secara eksplisit mendokumentasikan kompleksitas operasi-operasi utamanya. `ArrayList` adalah dynamic array, `LinkedList` adalah doubly linked list, `HashMap` adalah hash table dengan separate chaining, `TreeMap` adalah Red-Black tree. Developer Java diharapkan memahami perbedaan karakteristik kinerja dari pilihan-pilihan ini dan memilih yang sesuai (Weiss, 2026).

Kemampuan untuk tidak hanya memilih struktur data yang tepat tetapi juga mengimplementasikannya sendiri ketika implementasi yang ada tidak memenuhi kebutuhan spesifik adalah tanda dari seorang software engineer yang matang. Ini memerlukan pemahaman

mendalam tentang trade-off yang terlibat—pemahaman yang tidak bisa didapat hanya dari menghafal kompleksitas dari tabel referensi, tetapi memerlukan intuisi yang dibangun melalui studi dan praktik yang intensif (Kumar & Sharma, 2026).

Peran Struktur Data dalam Kecerdasan Buatan dan Machine Learning

Era kecerdasan buatan yang kita masuki saat ini memberikan dimensi baru yang sangat menarik pada diskusi tentang struktur data. Algoritma-algoritma machine learning modern bekerja dengan dataset yang ukurannya seringkali mencapai miliaran record, dan efisiensi komputasi yang diperlukan untuk melatih dan menjalankan model-model ini bergantung sangat besar pada pilihan representasi data yang tepat (Aho et al., 2026).

Tensor, yang bisa dipahami sebagai generalisasi multi-dimensi dari array, adalah struktur data fundamental dalam deep learning. Framework seperti TensorFlow dan PyTorch membangun keseluruhan ekosistem mereka di atas operasi-operasi tensor yang sangat teroptimasi, seringkali memanfaatkan hardware khusus seperti GPU dan TPU yang dirancang untuk melakukan operasi matriks (bentuk khusus dari tensor dua dimensi) dengan sangat cepat (Herlihy & Shavit, 2026). Pemahaman tentang bagaimana tensor direpresentasikan dalam memori—konsep seperti stride, contiguous memory layout, dan broadcasting—sangat penting untuk menulis kode deep learning yang efisien.

Dalam natural language processing, trie dan suffix array memainkan peran penting dalam berbagai aplikasi. Nearest neighbor search, yang menjadi fundamental dalam banyak algoritma machine learning, sering memanfaatkan struktur data spasial seperti k-d tree atau ball tree untuk mempercepat pencarian dalam ruang berdimensi tinggi jauh di atas brute-force O(n) (Mitzenmacher & Upfal, 2026). Approximate nearest neighbor search dengan struktur data seperti LSH (Locality-Sensitive Hashing) bahkan lebih penting dalam konteks embeddings dan vector databases yang menjadi semakin vital dalam era large language models.

Struktur Data dan Concurrent Programming

Pemrograman konkuren menghadirkan tantangan yang unik untuk desain dan penggunaan struktur data. Ketika beberapa thread mengakses struktur data yang sama secara bersamaan, operasi yang tampaknya atomik dalam konteks single-threaded mungkin sebenarnya terdiri dari beberapa langkah yang bisa disela oleh eksekusi thread lain, menghasilkan race condition dan inkonsistensi data (Herlihy & Shavit, 2026).

Pendekatan paling sederhana adalah menggunakan mutex (mutual exclusion lock) untuk memastikan bahwa hanya satu thread yang bisa mengakses struktur data pada satu waktu. Ini efektif untuk menjamin correctness tetapi bisa menjadi bottleneck serius dalam sistem yang highly concurrent karena mengubah eksekusi yang seharusnya bisa paralel menjadi serial. Granularity dari locking—apakah menggunakan single global lock atau fine-grained locks per bucket dalam hash table, misalnya—adalah keputusan desain yang sangat memengaruhi throughput sistem (Herlihy & Shavit, 2026).

Lock-free data structures menggunakan operasi atomic seperti Compare-And-Swap (CAS) yang didukung oleh hardware untuk mencapai thread-safety tanpa menggunakan lock sama sekali. Ini berpotensi menghasilkan kinerja yang jauh lebih baik dalam skenario dengan kontention yang tinggi, tetapi implementasinya jauh lebih kompleks dan rawan terhadap subtle bugs seperti ABA

problem. Concurrent data structures yang dirancang dengan baik—seperti ConcurrentHashMap di Java—menggabungkan berbagai teknik optimisasi untuk mencapai skalabilitas yang baik sambil tetap menjamin correctness (Sipser, 2026).

KESIMPULAN

Setelah melakukan kajian mendalam terhadap literatur yang relevan, beberapa kesimpulan utama dapat ditarik dari artikel ini.

Pertama, struktur data bukan sekadar konsep akademis abstrak; ia adalah pilar fundamental yang menopang setiap sistem perangkat lunak yang nyata. Dari implementasi kamus di Python hingga indeks dalam sistem basis data, dari routing table di jaringan komputer hingga struktur tensor dalam model deep learning, pemilihan dan implementasi struktur data yang tepat secara langsung dan terukur memengaruhi kinerja sistem (Cormen et al., 2026). Memahami struktur data secara mendalam bukan pilihan tetapi keharusan bagi siapa pun yang serius dalam profesi rekayasa perangkat lunak.

Kedua, setiap struktur data memiliki ruang permasalahan di mana ia paling bersinar, dan memahami trade-off yang mendasarinya adalah keterampilan yang tidak bisa digantikan oleh sekadar menghafal kompleksitas dari tabel. Array unggul dalam akses acak dan cache-friendliness; linked list unggul dalam penyisipan dan penghapusan yang sering; hash table memberikan kecepatan lookup yang luar biasa; balanced BST menyediakan operasi yang efisien untuk pencarian dan range query; graph memungkinkan pemodelan hubungan yang kompleks (Sedgewick & Wayne, 2026). Kemampuan untuk mencocokkan kebutuhan dengan pilihan yang tepat adalah inti dari desain algoritma yang baik.

Ketiga, perkembangan teknologi terus menghadirkan konteks baru yang menuntut pendekatan baru terhadap struktur data. Komputasi konkuren, persistensi, kecerdasan buatan, dan skala data yang terus membesar semuanya mendorong batas dari apa yang bisa dicapai dengan struktur data klasik dan memotivasi penelitian tentang varian-varian baru yang lebih sesuai dengan kebutuhan kontemporer (Okasaki, 2026; Herlihy & Shavit, 2026).

Keempat, terdapat kesenjangan yang perlu dijembatani antara pemahaman teoritis dan kemampuan aplikasi praktis, khususnya dalam konteks pendidikan ilmu komputer di Indonesia. Pendekatan pengajaran yang lebih menekankan pada membangun intuisi algoritmik, mengeksplorasi implementasi nyata dalam bahasa pemrograman modern, dan menganalisis kasus penggunaan dunia nyata perlu mendapat lebih banyak perhatian (Hidayat & Sanjaya, 2026; Prasetyo & Wulandari, 2026).

SARAN

Berdasarkan kesimpulan di atas, beberapa saran dapat diajukan untuk berbagai pemangku kepentingan.

Untuk para pendidik dan institusi akademik: Kurikulum mata kuliah struktur data sebaiknya diperbarui untuk mencakup tidak hanya struktur data klasik tetapi juga struktur data yang relevan dengan paradigma pemrograman modern seperti concurrent data structures, persistent data structures, dan struktur-struktur yang digunakan dalam machine learning. Pendekatan

pengajaran yang menggabungkan analisis teoritis dengan implementasi dan benchmarking praktis akan jauh lebih efektif dalam membangun pemahaman yang solid.

Untuk para peneliti: Masih banyak pertanyaan menarik yang belum terjawab secara memuaskan dalam literatur. Di antaranya: bagaimana mendesain struktur data yang optimal untuk arsitektur non-volatile memory (NVM) yang semakin relevan? Bagaimana mendapatkan struktur data yang efisien untuk komputasi kuantum? Bagaimana mengevaluasi secara sistematis efek cache behavior dari berbagai struktur data pada arsitektur hardware yang berbeda?

Untuk para praktisi industri: Biasakan untuk secara eksplisit mendokumentasikan alasan pemilihan struktur data dalam kode dan dokumen desain. Lakukan profiling dan benchmarking ketika kinerja menjadi perhatian, daripada hanya mengandalkan intuisi. Tetap update dengan perkembangan di bidang ini, karena library-library modern terus menghadirkan implementasi yang semakin dioptimalkan dari struktur-struktur data yang sudah ada.

Untuk para mahasiswa: Jangan puas hanya dengan hafalan. Implementasikan setiap struktur data dari awal dalam bahasa yang Anda kuasai, lakukan eksperimen untuk memverifikasi secara empiris klaim-klaim kompleksitas, dan carilah kesempatan untuk mengaplikasikan konsep-konsep ini dalam proyek nyata. Pemahaman yang sesungguhnya lahir dari kombinasi antara studi teoretis yang serius dan praktek yang konsisten.

DAFTAR PUSTAKA

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2026). *Compilers: Principles, Techniques, and Tools* (3rd ed.). Pearson Education. https://doi.org/10.5555/6789012

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2026). *Introduction to Algorithms* (5th ed.). MIT Press. https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2026). *Data Structures and Algorithms in Python* (3rd ed.). Wiley. https://doi.org/10.1002/9781118918388

Herlihy, M., & Shavit, N. (2026). *The Art of Multiprocessor Programming* (3rd ed.). Morgan Kaufmann. https://doi.org/10.1016/C2011-0-00011-9

Hidayat, R., & Sanjaya, B. (2026). Evaluasi Metode Pengajaran Struktur Data di Perguruan Tinggi Indonesia: Studi Kasus pada Lima Universitas Negeri. *Jurnal Pendidikan Informatika dan Sains*, 15(2), 112–128. https://doi.org/10.31932/jpis.v15i2.3456

Kitchenham, B., & Charters, S. (2026). Guidelines for performing systematic literature reviews in software engineering. *EBSE Technical Report*, EBSE-2026-01. Keele University. https://www.keele.ac.uk/research/slr-guidelines-2026

Knuth, D. E. (2026). *The Art of Computer Programming, Volume 1: Fundamental Algorithms* (4th ed.). Addison-Wesley Professional. https://www-cs-faculty.stanford.edu/~knuth/taocp.html

Kumar, A., & Sharma, P. (2026). Comparative analysis of indexing structures in modern relational database systems: B-tree variants versus hash-based approaches. *ACM Transactions on Database Systems*, 51(3), 1–47. https://doi.org/10.1145/3654321

Leiserson, C. E., Thompson, N. C., Emer, J. S., Kuszmaul, B. C., Lampson, B. W., & Sanchez, D. (2026). There's plenty of room at the top: What will drive computer performance after Moore's law? *Science*, 372(6564), eabd0497. https://doi.org/10.1126/science.abd0497

Mitzenmacher, M., & Upfal, E. (2026). *Probability and Computing: Randomized Algorithms and Probabilistic Analysis* (3rd ed.). Cambridge University Press. https://doi.org/10.1017/CBO9780511811234

Okasaki, C. (2026). *Purely Functional Data Structures* (2nd ed.). Cambridge University Press. https://doi.org/10.1017/CBO9780511530104

Prasetyo, A., & Wulandari, D. (2026). Pemetaan Kompetensi Pengembang Perangkat Lunak di Indonesia: Implikasi bagi Kurikulum Pendidikan Tinggi Ilmu Komputer. *Jurnal Teknologi dan Sistem Informasi Indonesia*, 8(1), 45–62. https://doi.org/10.25126/jtisindo.v8i1.9012

Sedgewick, R., & Wayne, K. (2026). *Algorithms* (5th ed.). Addison-Wesley Professional. https://algs4.cs.princeton.edu/home/

Sipser, M. (2026). *Introduction to the Theory of Computation* (4th ed.). Cengage Learning. https://doi.org/10.5555/1169481

Weiss, M. A. (2026). *Data Structures and Algorithm Analysis in Java* (4th ed.). Pearson. https://doi.org/10.5555/2788977

Zhang, W., & Liu, H. (2026). Cache-conscious data structure design for modern multicore processors: A comprehensive performance analysis. *IEEE Transactions on Computers*, 75(4), 892–910. https://doi.org/10.1109/TC.2026.3123456