Is It Time to Remove Data Structures? A Critical Look at Requirements and Curricular Placement.

Albert Lionelle
Khoury College of Computer Sciences
Northeastern University
Boston, MA, USA
a.lionelle@northeastern.edu

Abstract

CS 2: Data Structures is arguably one of the most central courses in a Computer Science degree, but how it is taught and the exact topics are often an area of debate among educators.

Indeed, a foundational tension has emerged with, on the one hand, students questioning the relevance of implementing algorithms readily available in modern software development kits (SDKs) and, on the other hand, faculty maintaining that deep algorithmic understanding remains essential for advanced coursework. By using topics identified in previous research and cross-referencing them with the ACM 2023 Knowledge Areas, the analysis reveals that current data structures courses attempt to serve two distinct educational objectives: first, to provide practical software development; and second, to ensure core theoretical algorithmic understanding. As such, the course is often overburdened with content and overwhelming to students. Additionally, by examining the curricular complexity of 75 degree programs, we show the placement of the course as highly central, creating complex bottlenecks for students compounding the difficulty of the course.

We argue for the benefits of replacing the traditional single course with two focused courses: (1) Object-Oriented Programming with Data Structures, emphasizing practical application through SDK usage and software engineering principles; and (2) Data Structures and Algorithms, providing mathematical foundations through experimental analysis and formal proofs. Sample topic lists mapped to Knowledge Areas are provided with each recommendation. While not unknown, this approach reduces curricular complexity while better serving diverse student populations, from those seeking practical programming skills to those requiring theoretical depth for advanced computer science study.

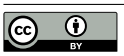
CCS Concepts

• **Social and professional topics → Model curricula**; **Computing education programs**; **Computing education**.

Keywords

CS 2, Data Structures, Curriculum, Curricular Complexity

ACM Reference Format:
Albert Lionelle. 2026. Is It Time to Remove Data Structures? A Critical Look at Requirements and Curricular Placement.. In *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V.1 (SIGCSE TS 2026), February 18–21, 2026, St. Louis, MO, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3770762.3772552



SIGCSE TS 2026, St. Louis, MO, USA

ACM ISBN 979-8-4007-2256-1/26/02
https://doi.org/10.1145/3770762.3772552

1 Introduction

During the SIGCSE Technical Symposium 2025 plenary session, award recipients participated in a panel discussing "The Future of CS Education [10]." A topic emerged when panelists characterized CS 2: Data Structures as outdated, with some suggesting it was time to eliminate the course in its current form. This assessment resonated with ongoing faculty concerns about the course [5], generating mixed reactions from the academic audience. Proponents of reform argued that requiring students to implement sorting algorithms seems disconnected from professional practice, where developers utilize built-in library functions. Counter-arguments emphasized that learning to write sorting algorithms serves a deeper pedagogical purpose: developing skills in algorithmic comparison, algorithm construction, and formal correctness analysis. This debate fundamentally questions whether students need to *use* data structures or truly *grok* them [3].

Both perspectives contain valid elements. Contemporary industry practices favor utilizing software development kits over custom implementations, making traditional implementation requirements appear antiquated. Conversely, students require foundational competencies in data structures with a deeper understanding of the algorithms to succeed in constructing and comprehending complex algorithms, both in advanced coursework and professional contexts. The deeper theoretical understanding of algorithmic principles and their appropriate applications arguably distinguishes computer scientists from self-taught programmers.

This tension raises a fundamental curricular question: does CS 2: Data Structures effectively achieve its intended learning objectives? The course originated during an era when computer scientists lacked comprehensive libraries, making implementation skills essential. While industry practices have evolved significantly, core competencies in algorithmic correctness and complex algorithm analysis remain crucial for advanced academic work. Furthermore, the course's perceived importance has positioned it centrally within most curricular structures, creating additional pedagogical and structural challenges that warrant examination.

To deepen our understanding of CS 2: Data Structures, we examine existing work on CS 2: Data Structures topics and issues in Section 2. Through analysis of 75 distinct degree programs and synthesis of commonly identified topics from prior research, we cross-reference curricular content with the 2023 ACM Knowledge

Areas while evaluating the structural complexity of course placement within Section 3. In Section 4, we propose a two-course framework to replace the traditional single course Data Structures model, and end with concluding remarks in Section 5.

2 Related Work

Computer science education research has extensively examined the introductory course sequence, typically comprised of CS 1, CS 2, Discrete Structures, Computer Organization, and Software Engineering (sometimes termed CS 3), though clear definitional boundaries remain elusive and most scholarly attention has concentrated on CS 0 and CS 1. Despite over a decade of CS 1 research, content definitions remain ambiguous. Responding to Becker's challenge [2], Mason et al. [18] conducted a global survey of introductory programming course content, building on prior studies that demonstrated that substantial disagreement exists regarding actual course content of CS 1 and CS 2 [13, 21].

CS 2 has earned a reputation as a notoriously challenging course among students [5]. Addressing this difficulty, Lionelle et al.[15] successfully reduced CS 2 failure rates by implementing a program that taught at-risk students effective learning strategies. When these same techniques were subsequently applied to CS 1, researchers observed improved CS 2 pass rates and enhanced student retention [16]. Investigating how CS 1 modifications might impact CS 2 performance, Gal-Ezer et al. [8] examined programming language choices in CS 1, finding that the introductory language has minimal effect on CS 2 student outcomes.

Roumani [20] proposed reorganizing CS 1 and CS 2 to establish clearer separation of concerns, though this approach concentrated CS 2 exclusively on Object-Oriented Programming. Conversely, Garber et al. [7] assumed a more theory-oriented CS 2 Data Structures course when demonstrating benefits of structured exam questions. Through examination of assessment outcomes, Danielsiek et al.[6] identified trees and heaps as particularly challenging topics for students, alongside conceptual difficulties distinguishing between divide-and-conquer and dynamic programming paradigms.

Porter et al. [19] focused on developing course level learning outcomes for CS 2. Surveying experts in the topic grouped common topics (documented by Hertz [13]) into **Object-Oriented programming** (class design, inheritance, polymorphism), **Basic Data Structures** (stack, queue, array, linked list, binary trees), **Recursion**, **Sorting**, **Algorithm Analysis**, and **Advanced Data Structures** (balanced trees, hash tables, heaps, priority queues, graphs). Even with their expert survey, there was little agreement on the topics and the level of detail for each topic. From this grouping of topics, they developed six course outcomes that blend software development and algorithmic analysis. Due to the variability of topics in CS 2, we use Porter's topic groupings as the basis for our definition of CS 2: Data Structures content.

3 Data Structures: Single Course Approach

To assess the breadth of what students are learning in current data structures courses, we cross-referenced the Knowledge Areas (KA) presented in the ACM Computer Science Curricula 2023 [14] with the topics presented in Porter et al. [19]. In Table 1, we pair each topic with every KA that mentions that topic. While there is no consistency on the depth of each topic, the KAs covered in a "standard" Data Structures course are: FPL-OOP, FPL-Functional (recursion only), SDF-Practices, SDF-Construction, SDF-Data Structures, SDF-Algorithms, AL-Foundational, AL-Complexity, AL- Models, AL-Strategies, and MSF-Discrete. All of these topics fall into the CS Core areas.

It is often assumed that MSF-Discrete is covered in a Discrete Math course, so, if we remove those topics, we are left with six topics focusing on software development, and four topics focused on algorithmic analysis. While it is encouraged that courses cross knowledge areas, and that institutions don't feel limited to the knowledge areas, presenting too much information in one course can have consequences for students' cognitive load. This distribution reveals two distinct educational objectives embedded within most data structure courses: solid software development principles and deep algorithmic understanding.

The software development focus is unsurprising, as data structure courses frequently involve determining which data structure is best applied given the context of a larger application. This analysis proves essential during technical interviews and forms the foundation for effective software development. Simultaneously, students are expected to transcend mere memorization of Big O complexities, instead developing the ability to prove correctness and understand fundamental differences between algorithmic design and implementation. This dual expectation naturally raises the pedagogical question: what should we teach first?

Traditional course designs cover one data structure or algorithm comprehensively before advancing to the next, while spiral approaches have students first work with libraries to see data structures applied in practice, then return for in-depth analysis of each structure. Both methodologies aim to strengthen students' programming and algorithmic understanding. However, students frequently struggle to retain all information, questioning, for example, the practical necessity of implementing algorithms like merge-sort when they may never encounter such requirements in practice. While there is more overlap with the outcomes (by design), this fundamental division persists between the **use of data structures** and **the algorithmic design of data structures**, suggesting complementary, yet distinct educational goals.

3.1 Curricular Placement

To enhance the field's consideration of the placement of Data Structures within the curriculum, we can apply the measures for curricular complexity developed by Heileman et al [11]. Curricular complexity represents a growing research area within computer science education [9, 12, 17], with course **centrality** serving as a key metric for evaluating a course's importance within curricular design. Centrality is quantified by measuring the number of prerequisite chains that traverse through a particular course. Courses with high centrality often function as "choke points" because when students encounter difficulties in these courses their progress toward graduation becomes blocked or severely constrained. This does not suggest that highly central courses are unnecessary, but rather that such courses require *enhanced support structures*, *reduced class sizes*, and *improved mechanisms for student comprehension.*

Lionelle et al. conducted a random sampling of 60 Computer Science programs, building upon Heileman's findings of an inverse

Table 1: CS2023 Topics and Knowledge Areas

Topic	Knowledge Area	Short Description
OOP	FPL-OOP	Foundations of programming languages includes object-oriented design, class inheritance, and polymorphism
	SDF-Practices	Write tests and debug objects; basic testing including test case design
	SE-Construction	Focuses on unit testing, documentation, and practical small-scale testing
Basic Data Structures (stacks, queues, arrays, linked lists, binary trees)	SDF-Data-Structures	Standard abstract data types (lists, stacks, queues, sets, maps/dictionaries), selecting appropriate data structures, performance implications
	AL-Foundational	Abstract Data Types (ADTs), arrays, linked lists, stacks, queues, hash tables, trees (binary, n-ary, search trees), sets, basic operations and algorithms
	AL-Complexity	Big O analysis, complexity classes (constant, logarithmic, linear, quadratic) for data structure operations
	AL-Strategies	Algorithmic paradigms demonstrated through data structures (brute force, divide-and-conquer, time-space tradeoffs)
	MSF-Discrete	Mathematical foundations: sets, relations, functions, basic graph theory, trees as mathematical structures
	FPL-OOP	Records/structs/tuples and objects as compound data types; relationship to object-oriented programming
Recursion	SDF-Fundamentals	Basic recursion concept, developing recursive functions, explain when and how to use recursion effectively
	AL-Strategies	Iteration vs recursion comparison, recursive algorithmic paradigms, recursive algorithms for tree/graph traversal
	AL-Models	Algorithmic correctness with invariants in recursion, formal verification of recursive algorithms
	MSF-Discrete	Recursive mathematical definitions, recurrence relations, inductive proofs, structural induction connections to recursion
	FPL-Functional	Effect-free recursive programming, recursion vs loops, processing structured data recursively, tail call optimization
Sorting (quadratic sorts: bubble, insertion, selection; divide & conquer: merge, quicksort)	SDF-Algorithms	Common sorting algorithms as fundamental algorithmic concepts, impact on time-space efficiency of programs
	AL-Foundational	Specific sorting algorithms: $O(n^2)$ complexity (insertion, selection), O(n log n) complexity (quicksort, merge, timsort)
	AL-Complexity	Complexity analysis of sorting algorithms: quadratic $O(n^2)$ vs log-linear O(n log n) complexity classes
	AL-Strategies	Algorithmic paradigms demonstrated through sorting: brute-force (selection sort), divide-and-conquer (mergesort, quicksort)
Algorithm Analysis (Big O, time/space complexity, best/worst case)	SDF-Algorithms	Basic concept of algorithm efficiency and notion of algorithm efficiency, impact of algorithms on time-space efficiency of programs
	AL-Complexity	Core complexity analysis framework: Big O, Omega, Theta notations; foundational complexity classes; best/average/worst-case analysis
	AL-Foundational	Complexity analysis applied to specific algorithms and data structures with concrete examples of different complexity classes
	AL-Strategies	Time-space tradeoffs as algorithmic design strategy; understanding how different paradigms impact complexity
Advanced Data Structures (balanced trees, hash tables, heaps, priority queues, graphs)	AL-Foundational	Core coverage: hash tables with collision resolution, balanced trees (AVL, Red-Black, Heap), graphs with representation and algorithms, priority queues
	AL-Complexity	Complexity analysis of advanced operations: logarithmic tree operations, constant hash table access, graph algorithm complexities
	AL-Strategies	Advanced algorithmic strategies: transform-and-conquer (heapsort), greedy algorithms (Dijkstra's, Kruskal's), dynamic programming
	MSF-Discrete	Mathematical foundations: graph theory basics, tree properties, set theory underlying hash tables and priority queues
	SDF-Data-Structures	Performance implications of advanced data structure choices, selecting appropriate structures for complex problems

Table 2: Course Distribution Analysis With Courses That Appear More Than 3 Times Across 75 Schools

Course	Count	Percent
CS 2: Data Structures	45	60.0%
CS 1	12	16.0%
Mathematics	8	10.7%
Object-Oriented Programming	4	5.3%
Systems	3	4.0%

relationship between program rankings (as defined by CSRankings.org) and structural complexity, where higher-ranked institutions demonstrated less complex curricular designs [12]. Lionelle et al. [17] discovered that institutions with higher representation of women graduating from their programs exhibited lower structural complexity, thereby presenting fewer systemic barriers for students. A fundamental component of this complexity involves prerequisite pathways and the influence of highly central courses on degree structures. When students struggle in high centrality courses, they frequently conclude that the major is unsuitable for them, leading to increased attrition, especially from under represented students.

Using the open data provided by Lionelle et al.[1], we analyzed the top centrality courses across seventy-five randomly sampled institutions, with results presented in Table 2. For each degree map, we identified the course (or courses in cases of ties) with the highest centrality and grouped them by similar names. Data Structures emerged as the clear leader, appearing as the most central course in 60% of the programs examined. This finding is particularly concerning given that Data Structures is also recognized as a course characterized by high enrollments and significant difficulty.

Figure 1 illustrates a common prerequisite structure that consistently appeared across the degree maps in our analysis. Furthermore, Brodley et al. [4] documented that among 199 CS degree programs, 110 (55%) required Calculus as either a co-requisite or prerequisite to Discrete Mathematics or CS 1. This mathematical requirement pattern is not uncommon and introduces additional prerequisites before CS 2, thereby increasing both the overall structural complexity of the degree and the centrality of CS 2 within the curricular framework.

Given the extensive coverage of Data Structures within the curriculum and its central role connecting foundational and advanced courses, Data Structures emerges as the pivotal course in CS programs. However, the course presents significant pedagogical challenges for both students and faculty [5], due to the substantial breadth of KAs that are compressed into a single semester. If computer science educators are committed to meaningful curricular design, we must critically examine whether we are overburdening students within this singular course. If our goal is to develop students who are both competent software engineers and possess a deep understanding of algorithmic design principles, the solution is not to eliminate Data Structures but rather to thoughtfully divide it into two distinct courses that allow for more focused and thorough coverage of these essential topics.

4 Data Structures: Two-Course Approach

A solution to the challenge highlighted by this analysis is to break Data Structures into two separate courses. One course would focus on software development principles that use data structures in an applied manner, while the other course would focus on rigorous algorithmic design of data structures and a deeper understanding of the implementations. While this approach isn't unknown, it isn't very common. Additionally, the few schools that do implement this approach use a structure in which one course is a prerequisite for the other. For example, they have CS 1 → OOP → Data Structures as their introductory sequence. All three courses focus on implementation. This tightly-coupled pairing still creates a "bottleneck" and increases curricular complexity adding unneeded difficulty for students. As such, our proposal is rather to have two courses that are not forced together via prerequisites, but that stand independent of each other structurally. This should not be confused with students not needing each course, but instead the understanding that students gain the knowledge they need by taking both through a flexible, but natural progression of degree requirements.

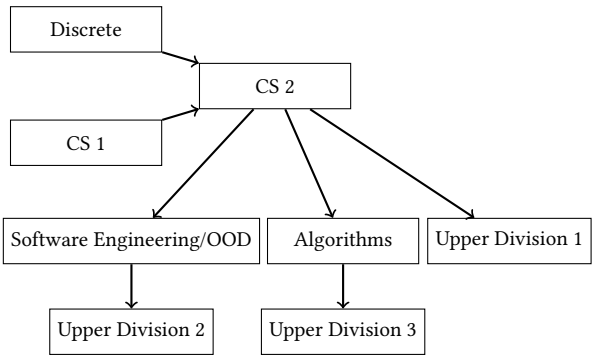


Figure 1: Sample Prerequisites with CS 2 Highly Central

It is worth noting that courses will vary based on the length of the university's semester and the exact topics the department wants to cover. The following is therefore meant as guidelines for course design.

4.1 Object Oriented Programming with Data Structures

The objective of an Object-Oriented Programming (OOP) with Data Structures course is to integrate software engineering principles early in the curriculum while leveraging existing SDK implementations of data structures. This approach allows students to focus on software design and architectural patterns rather than low-level implementation details. Students taking the course can still gain proficiency in Big O analysis and data structure comparisons, but through practical applications of swapping data structures to observe their impact on overall program performance and design decisions.

By incorporating design patterns directly into the OOP curriculum, the course allows later courses to introduce more advanced topics (e.g. Human-Computer Interaction) earlier in the student's academic progression. Table 3 presents fourteen potential topics for this course, assuming students enter with only procedural programming experience. Students with early objects exposure from CS 1 would benefit from the opportunity to explore advanced design principles.

[1] https://github.com/NeuCurricularAnalytics/DegreeMaps

Table 3: OOP Course Topics and Associated Knowledge Areas

Topic	Knowledge Areas
Introduction to Object-Oriented Programming	SDF-Fundamentals, FPL-OOP
Constructors and Method Overloading	SDF-Fundamentals, FPL-OOP
Unit Testing and Test-Driven Development	SDF-Practices, SE-Construction
Encapsulation and Data Hiding	FPL-OOP, SDF-Fundamentals
Inheritance Fundamentals	FPL-OOP
Advanced Inheritance and Polymorphism	FPL-OOP
Interfaces and Multiple Inheritance	FPL-OOP
Exception Handling and Debugging	SDF-Fundamentals, SDF-Practices
Introduction to Collections and Lists	SDF-Data-Structures, AL-Foundational
Advanced Generics and Type Safety	FPL-Types, SDF-Data-Structures
Sorted Structures - Trees	SDF-Data-Structures, AL-Foundational
Associative Structures - Hash-based Collections	SDF-Data-Structures, AL-Foundational
Streams and Functional Programming	FPL-Functional, SDF-Algorithms
Advanced Testing and Code Quality	SDF-Practices, SE-Construction

Beyond the core fourteen-week curriculum, professional skills could be integrated based on program needs, such as:

- Design-to-implementation workflows (lightweight UML, working within larger system architectures)
- Source control management
- Collaborative programming practices
- Code reviews and technical communication skills

A key advantage of OOP with Data Structures is that it requires only introductory programming experience. Because of the minimal prerequisites, the course needs an intentional and focused introduction to algorithmic analysis, specifically Big O notation for comparing algorithmic efficiency. In an era where Large Language Models increasingly influence programming practices, the proposed course emphasizes design thinking, comprehensive testing, and design evaluation within industry-standard development environments. Students engage with larger-scale projects while utilizing professionally implemented tools and frameworks, preparing them for real-world software development challenges that extend beyond individual algorithmic implementation.

4.2 Data Structures and Algorithms

The Data Structures and Algorithms course would address the mathematical rigor and analytical thinking essential for advanced computer science coursework in theory and more mathematically-advanced topics (e.g. AI/ML, Compilers). Our proposed approach, detailed in Table 4, establishes foundational understanding by beginning with computer memory architecture and its relationship to pointers and references. Students then progress to the generalized RAM computational model while developing competency in comprehensive algorithmic analysis tools, including nuanced understanding of Big O, Big Theta, and Big Omega notations.

Table 4: Data Structures and Algorithms Course Topics and Associated Knowledge Areas

Topic	Knowledge Areas
Memory and System Fundamentals	SF-Memory, AR-Memory
Introduction to Algorithm Analysis	AL-Complexity, AL-Models
Mathematical Foundations for Proofs	MSF-Discrete
Quadratic Sorting Algorithms	AL-Foundational, AL-Complexity, MSF-Discrete
Divide and Conquer Introduction	AL-Strategies, AL-Complexity, MSF-Discrete
Advanced Divide and Conquer	AL-Strategies, AL-Complexity, MSF-Probability
Sequential Data Structures - Arrays and Lists	AL-Foundational, SDF-Data-Structures
Sequential Data Structures - Advanced Topics	AL-Foundational, SDF-Data-Structures, AR-Memory
Hash Tables and Associative Structures	AL-Foundational, SDF-Data-Structures
Tree Structures and Binary Search Trees	AL-Foundational, SDF-Data-Structures
Heaps and Priority Queues	AL-Foundational, SDF-Data-Structures
Graph Representations and Basic Algorithms	AL-Foundational, MSF-Discrete
Greedy Algorithms	AL-Strategies
Advanced Topics and Integration	AL-Complexity, MSF-Discrete

While implementation remains important, this course emphasizes pseudo code analysis to strengthen students' ability to translate abstract algorithmic descriptions into working implementations. The prerequisite structure entails Discrete Mathematics for mathematical foundations and CS 1 for procedural programming experience, ensuring students enter with appropriate preparation.

This course adopts an experiment-driven pedagogical approach where students conduct rigorous empirical analysis of algorithmic performance. Students develop competency in comparing theoretical runtime bounds against actual execution time, constructing comprehensive datasets from algorithm performance, creating visualizations of execution patterns, and validating theoretical predictions through empirical evidence. This experimental focus recognizes that while data structure implementation techniques are well documented (and easily implemented with LLMs), deep learning emerges from hands-on investigation of performance characteristics across varied dataset sizes.

Professional skills integrated in the course include:

- Technical writing through algorithm analysis reports and research documentation
- Data analysis and visualization tools for empirical performance studies

- Problem-solving methodologies with algorithms
- Technical interview preparation and communication
- Introduction to research methodologies and scientific inquiry

The research methods component culminates in a comprehensive final project in which students independently investigate a data structure not covered in the coursework. This project requires complete theoretical analysis, implementation, and empirical evaluation, developing critical skills for lifelong learning in computer science. Since comprehensive coverage of all data structures and algorithms is impossible within a single course, this research-based approach equips students with the analytical frameworks essential for future academic work and professional practice, enabling them to evaluate and compare any data structure or algorithm they encounter.

4.3 Curricular Mapping

The centrality of traditional Data Structures courses creates significant curricular bottlenecks that affect computer science majors but also impact students pursuing interdisciplinary degrees and also those looking to complete a CS minor. This structural constraint forces computer scientists to justify intensive algorithmic theory to students who mostly need practical programming skills. The two-course approach resolves this tension by providing separate pathways: one emphasizing software engineering fundamentals with practical data structure usage, and another focusing on algorithmic theory and mathematical rigor.

Figure 2 illustrates how the updated prerequisite structure creates clearer pathways compared to the highly connected model where Data Structures served as an almost universal bottleneck. The inclusion of College Algebra as a foundational requirement reflects contemporary curricular recommendations; while Calculus often appears as a prerequisite for Discrete Mathematics, multiple authoritative sources, including the ACM's 2023 Curricular Guidelines, advocate for repositioning Calculus later in the curriculum [1, 4, 14, 17]. This recommendation aligns with that guidance by establishing a domain specific mathematical foundation through an algorithmic focused Data Structures course, providing stronger preparation for computer science theory while updating the placement of Calculus in the degree and ensuring it not an unnecessary prerequisite barrier early on.

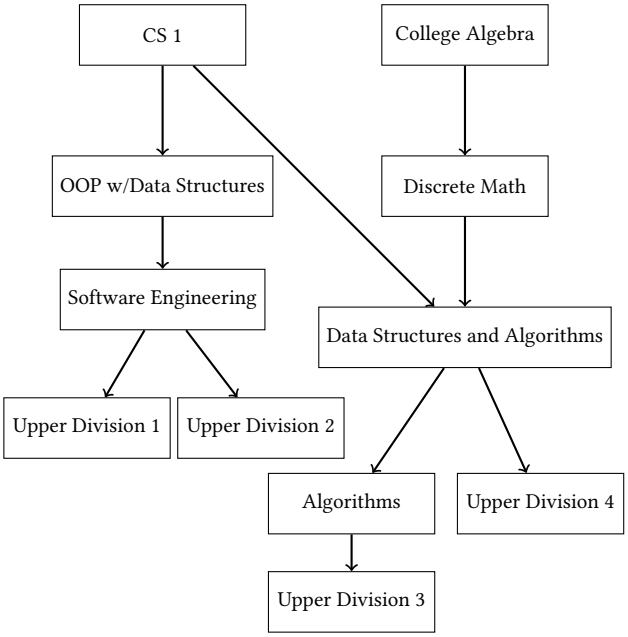


Figure 2: Updated Prerequisite Pathway

A caution about adding a course to CS degree plans: simply increasing the number of courses is not the recommended approach. Instead, our proposal should be part of an overall re-consideration regarding the flow and flexibility of the degree. In other words, this approach would be coupled with credit consolidation elsewhere.

The prerequisite structure presented here represents one possible configuration among many. Additional specialized pathways could branch directly from OOP with Data Structures, including domain-specific courses such as VR applications, mobile application development, and web development. Having multiple curricular pathways helps CS departments reduce systemic barriers that constrain student progress, limit program accessibility, and can result in dire consequences for some students.

5 Conclusions

Traditional Data Structures courses can be a source of frustrating tension between students and faculty, with students questioning the relevance of certain topics and faculty standing firmly behind the need for essential foundational skills. Further, the course's central placement in most degree plans creates curricular bottlenecks that constrain student progress.

Rather than overwhelming students with so much content in one single course, distributing knowledge across two focused courses should be an avenue of exploration. The first course would emphasize software development through object-oriented programming with SDK-based data structures, providing flexibility for subsequent courses and early professional skills. The second would focus on algorithmic theory and research methodology through empirical experiments, performance analysis, and formal proofs, preparing students for advanced theoretical coursework.

Dividing Data Structures into two specialized courses would enable institutions to optimize prerequisite structures and eliminate bottlenecks, benefiting both computer science majors and students in interdisciplinary programs. However, course additions require careful consideration, especially in the context of intensive degree frameworks. Students need increased flexibility, not additional requirements. Implementation of the recommendations should involve reassessing mathematics placement and evaluating courses with diminished relevance.

The goal is curricular optimization, not expansion: to create effective pathways that expand options and accessibility rather than limit flexibility and increase student burden. Data Structures is rightfully a challenging course, but we argue that, through thoughtful curricular revision, the course could be less overburdened, removed as a chronic bottleneck, and better aligned with student and faculty expectations. As a community, we must collectively reassess CS 2's role in fostering student success and retention.

Acknowledgments

We would like to thank Catherine Gill, Megan Giodano, Rasika Bhalerao, Alvaro Monge, and Heather Lionelle for early reviews.

References

[1] Douglas Baldwin, Amanda Holland-Minkley, and Grant Braught. 2019. Report of the SIGCSE committee on computing education in liberal arts colleges. *ACM Inroads* 10, 2 (4 2019), 22–29. doi:10.1145/3314027

[2] Brett A. Becker. 2021. What does saying that 'programming is hard' really say, and about whom? *Commun. ACM* 64, 8 (8 2021), 27–29. doi:10.1145/3469115

[3] Richard Blumenthal and Johanna Blumenthal. 2025. Moving What's in the CS Curriculum Forward A Proposition to Address Ten Wicked Curricular Issues. *SIGCSE TS 2025 - Proceedings of the 56th ACM Technical Symposium on Computer Science Education* 1 (2 2025), 137–143. doi:10.1145/3641554.3701924

[4] Carla E. Brodley, Mckenna Quam, and Mark Weiss. 2024. An Analysis of the Math Requirements of 199 CS BS/ BA Degrees at 158 U.S. Universities. *Commun. ACM* 67, 8 (7 2024), 122–131. doi:10.1145/3661482

[5] Victoria C. Chávez. 2023. How My Students and I (Re)Discovered the Joy of Computing in CS2. *ACM Inroads* 14, 2 (5 2023), 36–39. doi:10.1145/359691

[6] Holger Danielsiek, Wolfgang Paul, and Jan Vahrenhold. 2012. Detecting and understanding students' misconceptions related to algorithms and data structures. *SIGCSE'12 - Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (2012), 21–26. doi:10.1145/2157136.2157148

[7] Iris Gaber, Amir Kirsh, and David Statter. 2023. Studied Questions in Data Structures and Algorithms Assessments. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE* 1 (6 2023), 250–256. doi:10.1145/3587102.3588843

[8] Judith Gal-Ezer, Tamar Vilner, and Ela Zur. 2009. Has the paradigm shift in CS1 a harmful effect on data structures courses: A case study. *SIGCSE'09 - Proceedings of the 40th ACM Technical Symposium on Computer Science Education* (2009), 126–130. doi:10.1145/1508865.1508909

[9] Sumukhi Ganesan, Albert Lionelle, Catherine Gill, and Carla Brodley. 2025. Does Reducing Curricular Complexity Impact Student Success in Computer Science? *SIGCSE TS 2025 - Proceedings of the 56th ACM Technical Symposium on Computer Science Education* 1 (2 2025), 360–366. doi:10.1145/3641554.3701915

[10] Daniel Garcia (moderator), Mitchel Resnick, Manuel A. Pérez-Quiñones, and Jonathan Mwaura. 2025. Awards Presentation and Panel – "The Future of CS Education". In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education* (Pittsburgh, PA, USA) *(SIGCSE TS '25)*. Association for Computing Machinery, New York, NY, USA. Panel discussion.

[11] Gregory L. Heileman, Chaouki T. Abdallah, Ahmad Slim, and Michael Hickman. 2018. Curricular Analytics: A Framework for Quantifying the Impact of Curricular Reforms and Pedagogical Innovations. (11 2018). https://arxiv.org/abs/1811.09676v1

[12] Gregory L. Heileman, Hayden W. Free, Johnny Flynn, Camden Mackowiak, Jerzy W. Jaromczyk, and Chaouki T. Abdallah. 2020. Curricular Complexity Versus Quality of Computer Science Programs. (6 2020). https://arxiv.org/abs/2006.06761v1

[13] Matthew Hertz. 2010. What do "CS1" and "CS2" mean? Investigating differences in the early courses. *SIGCSE'10 - Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (2010), 199–203. doi:10.1145/1734263.1734335

[14] Amruth N Kumar, Rajendra K Raj, Sherif G Aly, Monica D Anderson, Brett A Becker, Richard L Blumenthal, Eric Eaton, Susan L Epstein, Michael Goldweber, Pankaj Jalote, Douglas Lea, Michael Oudshoorn, Marcelo Pias, Susan Reiser, Christian Servin, Rahul Simha, Titus Winters, and Qiao Xiang. 2024. *Computer Science Curricula 2023*. Association for Computing Machinery, New York, NY, USA. https://dl.acm.org/doi/pdf/10.1145/3664191

[15] Albert Lionelle, Sudipto Ghosh, Shannon Ourada, and Westin Musser. 2022. Increase Performance and Retention: Teach Students How To Study. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, USA, 349–355. doi:10.1145/3478431.3499340

[16] Albert Lionelle, Sudipto Ghosh, Benjamin Say, and J. Ross Beveridge. 2022. Increase Performance in CS 2 via a Spiral Redesign of CS 1. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, USA, 502–508. doi:10.1145/3478431.3499339

[17] Albert Lionelle, McKenna Quam, Carla Brodley, and Catherine Gill. 2024. Does Curricular Complexity in Computer Science Influence the Representation of Women CS Graduates?. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024)*. ACM, New York, NY, USA, 729–735. doi:10.1145/3626252.3630835

[18] Raina Mason, Simon, Brett A. Becker, Tom Crick, and James H. Davenport. 2024. A Global Survey of Introductory Programming Courses. *SIGCSE 2024 - Proceedings of the 55th ACM Technical Symposium on Computer Science Education* 1 (3 2024), 799–805. doi:10.1145/3626252.3630761

[19] Leo Porter, Daniel Zingaro, Cynthia Lee, Cynthia Taylor, Kevin C. Webb, and Michael Clancy. 2018. Developing course-Level learning goals for basic data structures in CS2. *SIGCSE 2018 - Proceedings of the 49th ACM Technical Symposium on Computer Science Education* 2018-Janua (2 2018), 858–863. doi:10.1145/3159450.3159457

[20] Hamzeh Roumani. 2007. Practice what you preach: Full separation of concerns in CS1/CS2. *Proceedings of the Thirty-Seventh SIGCSE Technical Symposium on Computer Science Education* (2007), 491–494. doi:10.1145/1121341.112149

[21] Carsten Schulte and Jens Bennedsen. 2006. What do teachers teach in introductory programming? *ICER 2006 - Proceedings of the 2nd International Computing Education Research Workshop* 2006 (2006), 17–28. doi:10.1145/1151588.1151593