**Lessons Learned Reflection: Data Structures and Algorithms**

Jeremy Carney

Colorado State University Global Campus

CSC400: Data Structures and Algorithms

Dr. Angelo Luo

July 6th, 2025

**Lessons Learned Reflection: Data Structures and Algorithms**

This Data Structures and Algorithms course has provided a strong foundation in key computer science concepts. Below, I summarize the main lessons learned and reflect on their application to effective coding, drawing from the course modules and my GitHub projects.

**Abstract Data Types and Bag Implementations**

Modules 1 and 2 introduced abstract data types (ADTs) and the bag ADT, implemented using fixed-size arrays, dynamically resizable arrays, and linked structures. Working on these implementations, such as in the Module 1 Critical Thinking assignment, taught me the importance of choosing appropriate data structures based on operational needs. Fixed-size arrays are simple but rigid, while dynamic arrays and linked implementations offer flexibility at the cost of complexity. This experience, reflected in my GitHub project for Module 2, showed me how to balance memory usage and performance, a skill I’ll apply to optimize data handling in future projects.

**Algorithm Efficiency and Big Oh Notation**

Module 3’s focus on algorithm efficiency and Big Oh notation was pivotal. Analyzing the efficiency of bag implementations clarified how time and space complexity impact performance. My Module 3 Critical Thinking project involved comparing array-based and linked bag operations, revealing the trade-offs in real-world scenarios. Understanding Big Oh helps me evaluate algorithms before coding, ensuring I select efficient solutions for tasks like data processing or search operations.

**Stacks and Their Applications**

Module 4 explored the stack ADT and its applications, such as evaluating algebraic expressions and managing program execution. Implementing a stack-based expression evaluator in the Module 4 Critical Thinking assignment deepened my appreciation for stacks in parsing and recursion. This knowledge is directly applicable to writing robust code for compilers or undo functionalities, where last-in-first-out behavior is critical.

**Recursion and Performance**

Module 5’s recursion concepts were challenging but rewarding. Writing recursive methods for tasks like factorial computation in the Module 5 Critical Thinking project taught me to break problems into smaller, manageable subproblems. However, evaluating recursive performance highlighted the risk of stack overflow for deep recursions. This lesson guides me to use iteration where recursion is inefficient, improving code reliability in large-scale applications.

**Sorting Algorithms**

Modules 6 and 7 covered sorting algorithms, from selection and insertion sorts to advanced ones like quicksort and radix sort. Implementing these in my GitHub projects, particularly the Shell sort in Module 6, showed how algorithm choice affects performance on different data sets. For example, quicksort’s efficiency on large, random data contrasts with insertion sort’s effectiveness on nearly sorted data. This understanding will inform my approach to optimizing data processing tasks, ensuring I select the right algorithm for the context.

**Queues, Deques, and Priority Queues**

Module 8 introduced queues, deques, and priority queues, which I implemented in the Module 8 Portfolio Project. These structures are essential for managing ordered data, such as task scheduling or event handling. My project involved a priority queue for a simulated job scheduler, reinforcing the need for efficient data retrieval. This experience will help me design systems that handle asynchronous tasks or prioritize critical operations effectively.

**Application to Effective Coding**

These lessons translate directly to better coding practices. First, selecting the right data structure—whether a stack for recursion or a priority queue for task management—reduces complexity and improves performance. My GitHub projects demonstrate this through practical implementations. Second, analyzing algorithm efficiency before coding, as practiced in Module 3, ensures scalable solutions. Finally, modular design, emphasized across assignments, promotes reusable and maintainable code. For instance, separating ADT logic from application code in my Module 8 project made debugging easier.

In summary, this course has equipped me with tools to write efficient, maintainable code. By applying these principles—choosing appropriate data structures, optimizing algorithms, and designing modular systems—I can tackle complex programming challenges with confidence.