

Assignment 6

Assignment 6 (Part 2)

Prashanna Acharya

Algorithms and Data Structures (MSCS-532-M80)

University of the Cumberlands

GitHub Link: <https://github.com/prashcodes10>

Implementation

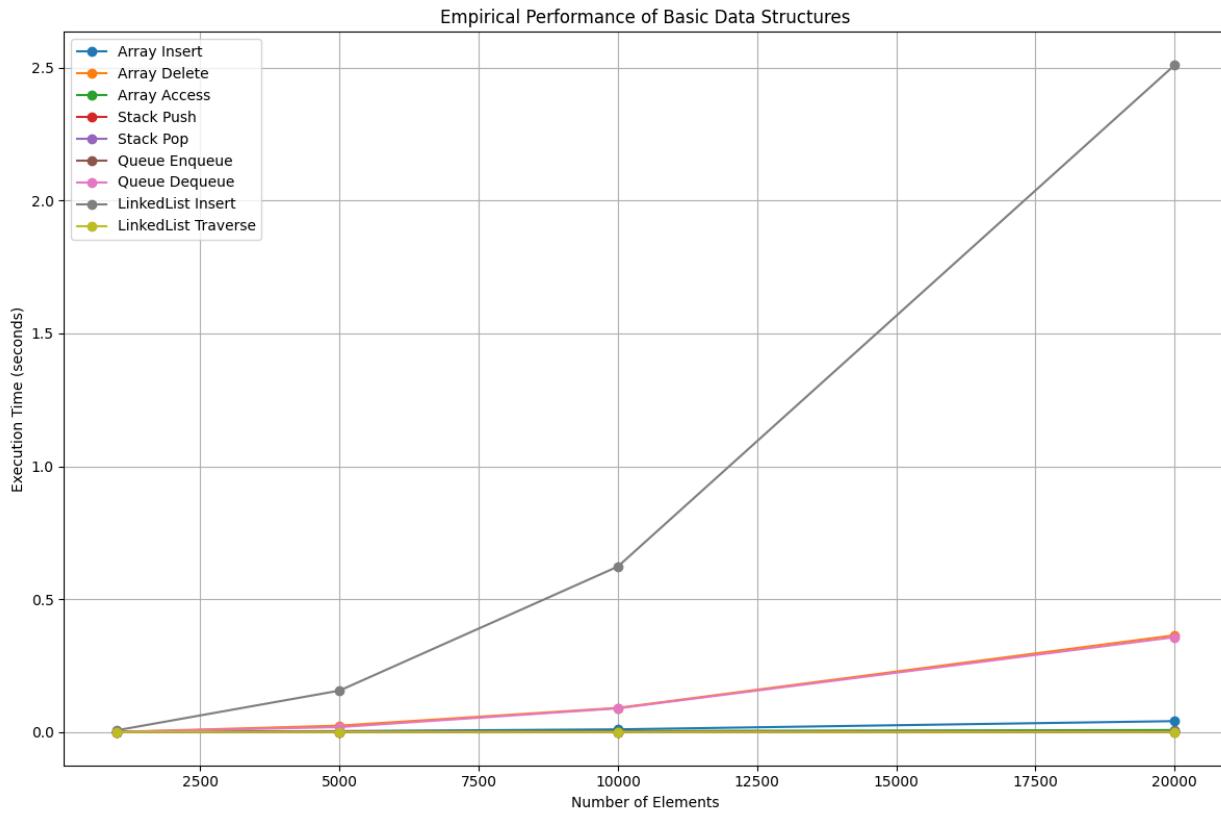
The below Python program implements five foundational data structures: arrays, matrices, stacks, queues, and singly linked lists. Each data structure is built from scratch without relying on Python's built-in data structures, demonstrating fundamental operations. Arrays support insertion, deletion, and access by index. Matrices extend arrays to two dimensions, enabling row and column manipulation. Stacks and queues are implemented using arrays, where stacks follow a last-in-first-out (LIFO) principle and queues follow a first-in-first-out (FIFO) principle. The singly linked list is built using nodes connected by pointers, supporting dynamic insertion and deletion at both the head and tail.

```

❸ Elementary Data Structures (Implementation & Analysis).py X
C: > Users > prash > Desktop > Masters > Fall 2025 > Algorithm and Data Structures > Assignment 6 > Assignment 6 - Part 2 > ❸ Elementary Data Structures (Implementation & Analysis).py >
38     def __init__(self):
39         self.items = []
40
41     def enqueue(self, value):
42         self.items.append(value)
43
44     def dequeue(self):
45         if self.items:
46             return self.items.pop(0)
47
48     class Node:
49         def __init__(self, value):
50             self.value = value
51             self.next = None
52
53     class LinkedList:
54         def __init__(self):
55             self.head = None
56
57         def insert_at_tail(self, value):
58             new_node = Node(value)
59             if not self.head:
60                 self.head = new_node
61                 return
62             current = self.head
63             while current.next:
64                 current = current.next
65             current.next = new_node
66
67         def delete_value(self, value):
68             current = self.head
69             prev = None
70             while current:
71                 if current.value == value:
72                     if prev:
73                         prev.next = current.next
74                     else:
75                         self.head = current.next
76                     return
77                 prev = current
78                 current = current.next
79
80         def traverse(self):
81             current = self.head
82             while current:

```

Performance Analysis



Arrays: Accessing elements in an array is $O(1)$ due to direct indexing. Insertion and deletion are $O(n)$ in the worst case because elements may need to be shifted to maintain order.

Matrices: Operations on a fixed-size matrix take $O(1)$ for single element access or modification. Iterating through the matrix is $O(n \times m)$, where n and m are the number of rows and columns.

Stacks and Queues (Array-based): Push, pop, enqueue, and dequeue operations are $O(1)$ on average for stacks and $O(n)$ for queue dequeue (since elements are shifted). Using a linked list for queues can reduce dequeue time to $O(1)$.

Linked Lists: Insertion and deletion at the head are $O(1)$, while insertion/deletion at the tail or arbitrary positions is $O(n)$ due to traversal. Access by index requires $O(n)$ time, making linked lists less efficient than arrays for random access.

The trade-offs highlight that arrays excel at fast random access but incur shifting costs for insertion/deletion, whereas linked lists provide dynamic memory usage with efficient head operations but slower indexed access.

Practical Applications

Arrays and Matrices are widely used in applications requiring fast indexing and multi-dimensional storage, such as image processing, scientific computing, and game development. Stacks are ideal for expression evaluation, backtracking algorithms, and undo functionalities. Queues are applied in task scheduling, breadth-first search, and real-time event handling. Linked lists are suitable for scenarios requiring frequent insertion and deletion, such as memory management, dynamic lists, and adjacency lists in graph representations.

Choosing the right data structure depends on the use case: arrays are preferred for speed in indexed access and compact memory storage, whereas linked lists are useful for dynamic, frequently changing datasets. Stacks and queues can be implemented with either arrays or linked lists depending on whether memory overhead or operation speed is more critical.

References

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.
- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data structures and algorithms in Python*. Wiley.
- Knuth, D. E. (1997). *The art of computer programming, Volume 1: Fundamental algorithms* (3rd ed.). Addison-Wesley.
- Weiss, M. A. (2013). *Data structures and algorithm analysis in Python* (2nd ed.). Pearson.
- Python Software Foundation. (2025). *Python documentation: Data structures*.
<https://docs.python.org/3/tutorial/datastructures.html>
- GeeksforGeeks. (2025). *Arrays, linked lists, stacks, and queues in Python*.
<https://www.geeksforgeeks.org>