

NAME: Famuyigun Olajide . I - 230502025  
 NATRIC: Computer Science  
 DEPT.: CSC 301 → Data Structures  
 Course: ASSIGNMENT

Questions → Exercise 5 -

1. Write short answers to:

- What is the difference between arrays and Linked List?
- What is the time complexity of insertion in a linked list?

5. Laboratory Work (To be done on the laptop)

6. Discussion Questions;

- What are the key differences between primitive data types and ADTs?
- Why are arrays considered static and linked list dynamic?
- In what situations would you prefer a linked list over an array?
- Give real-world examples where each of the following would be useful:
  - \* Stack
  - \* Queue
  - \* Linked List

Answers →

FEATURE	Array	Linked List
Memory storage	Contiguous	Non-Contiguous
Size	Fixed (Static)	Dynamic
Insertion/Deletion	$O(n)$ (Shifting required)	$O(1)$ if pointer is known
Access by index	$O(1)$ - Random access	$O(n)$ - Sequential access only
Memory Overhead	LOW	High (Stores pointer(s))

(ii) - Insert at beginning:  $O(1)$

- Insert at end:  $O(1)$  with tail pointer or  $O(n)$  (without tail pointer)

- Insert at a specific position:  $O(n)$  (to reach the position) +  $O(1)$  actual insertion

6.(i) Primitive Data Types	Abstract Data Types
* Built into the language (int, float, char, etc.)	User / programmer-defined

* Only store raw values * Fixed operation by language Implementation is visible Examples: int, double, boolean	Store data + define operations / behaviour Operations defined by the designer Implementation is hidden (abstraction) Examples: Stack, Queue, List, Map
---	---

- (b) \* Arrays are static because their size must be known and fixed at creation (in most languages). Once created, you cannot efficiently resize them.
- \* Linked lists are dynamic because nodes can be added or removed at runtime without re-allocating the entire structure. Memory is allocated individually for each node.

- (c)
- \* Frequent insertions/deletions at the beginning or middle
  - \* Size of the collection is ~~known~~<sup>unknown</sup> or changes drastically
  - \* Memory is fragmented and contiguous block is hard to allocate
  - \* You need efficient splitting/merging of lists
  - \* Implementing structures like stacks/queues where only ends are accessed.

(d) Data Structure		Real-World Examples
i.	Stack	<ul style="list-style-type: none"> <li>→ Undo/Redo feature in editors</li> <li>→ Browser back button</li> <li>→ Function call stacks</li> </ul>
ii.	Queue	<ul style="list-style-type: none"> <li>→ Printer job scheduling</li> <li>→ CPU task scheduling (round-robin)</li> <li>→ Customer service lines</li> </ul>
iii.	Linked List	<ul style="list-style-type: none"> <li>→ Music playlist (insert/delete songs easily)</li> <li>→ Image viewer (next/previous)</li> <li>→ Representing polynomials.</li> </ul>

#### Exercise 4:-

1) Analyzing time complexity;

Time Complexity:  $O(n)$

→ We visit each element exactly once.

Space Complexity:  $O(1)$

→ Only one extra variable total is used, regardless of input size.

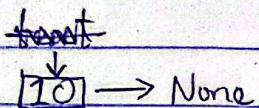
2) Trace how linked List insertion at the head works using diagrams.

Initial state (Before insertion):

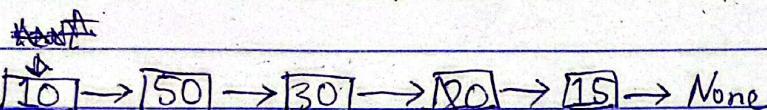
Head → [50] → [30] → [20] → [15] → null

Step-by-step insertion at Head:

(i) Create the new node:



(ii) Set the new node's next pointer to current head:



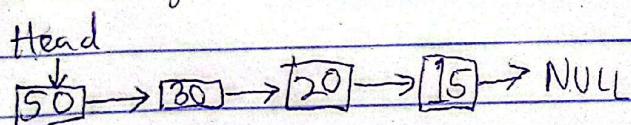
(iii) Update the head to point to the new node:

Head → [10] → [50] → [30] → [20] → [15] → None

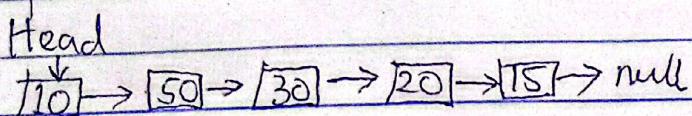
Final State (After Insertion):

Head → [10] → [50] → [30] → [20] → [15] → None

Visual Diagram (Before Insertion):



After insertion 10 at head:



Time complexity of insert at head: O(1)

Space complexity: O(1)