# Challenge 1 — Insert at the Front

Q: Insert a new node at the start of a linked list. What is the complexity?

+O(1) because we changed the head pointer to a new node.

Discuss: How is this easier compared to inserting at index 0 in an array?

+Arrays require shifting

# Challenge 2 — Insert at the End

Q: Append a new node to the end of a linked list. What is the complexity?

+ O(n) if we don’t include the end pointer.

Discuss: Do we need to traverse the entire list? How does this differ from arrays?

+Yes. In arrays, inserting a node at the end will always be O(1).

# Challenge 3 — Insert in the Middle

Q: Insert a node between two existing nodes.

Discuss: Which two arrows (pointers) need to be changed? Compare to shifting in arrays.

From “Previous Node” to “New Node” and from “New Node” to “Next Node”. Arrays mostly require shifting.

# Challenge 4 — Delete from the Front

Q: Remove the first node.

Discuss: What happens to the head pointer? What about the deleted node’s memory?

+It will move to the second node. The deleted node will be freed.

# Challenge 5 — Delete from the End

Q: Remove the last node.

Discuss: How do we find the node before the last one?

+Stop right before the last one and set its “nullptr”.

# Challenge 6 — Delete from the Middle

Q: Remove a node between two others.

Discuss: Which arrow changes? What happens if we forget to free memory?  
+Previous node. If we forgot to free memory, it will cause a memory leak.

# Challenge 7 — Traverse the List

Q: Print all elements in the linked list.

Discuss: How does traversal differ from direct arr[i] access?

traversal follows the pointers, whereas the array has full control over arr[i].

# Challenge 8 — Swap Two Nodes

Q: Swap two nodes in the list (not just their values).

Discuss: Is it easier to swap values or swap links? Why?

+swap values, because swapping links requires updating pointers, which can be more prone to a lot of errors.

# Challenge 9 — Search in Linked List

Q: Search for a value in a linked list.

Discuss: How is this similar to linear search in arrays? Which one is faster for random access?

both can gain access to their own arrays, but arrays is more faster then linked lists.

# Challenge 10 — Compare with Arrays

Q: For each operation (insert, delete, access), write the complexity for arrays vs. linked lists. Discuss: In what situations is a linked list clearly better?

+Linked list excels in insertions and deletion, either at the front or the middle.

# Reflection Prompts

1. Which operations were **O(1)** in linked lists but **O(n)** in arrays?
2. Which operation is clearly faster in arrays than in linked lists?
3. Why must we manage memory carefully in linked lists?
4. What does the **head pointer** represent?
5. What happens if we lose the head pointer?

# Scenario Analysis: Choose Array or Linked List

Read the following scenarios and decide whether an **array** or a **linked list** is a better fit. Justify your choice.

1. **Real-time scoreboard** where new scores are always added at the **end** and sometimes removed from the **front**.
2. **Undo/Redo feature in a text editor**, where operations are frequently added and removed at the **front**.
3. **Music playlist** that lets users add and remove songs anywhere in the list.
4. **Large dataset search** where random access by index is needed often.
5. **Simulation of a queue at a bank**, where customers join at the end and leave at the front.
6. **Inventory system** where you always know the item’s index and need quick lookups.
7. **Polynomial addition program** where terms are inserted and deleted dynamically. 8. **Student roll-call system** where the order is fixed and access by index is frequent.