Analyst questions

Challeng 1 Insert at the front

The complexity is O(n). if we compared to an arrays it more easier than using arrays cause we don't check all element.

```
void InsertFront(int value){
        Node* newNode = new Node{value, head};
        head = newNode;
        n++;
    }
```

Challenge 2 insert at end

Complexity of Inserting at the End

The time complexity for appending a new node to the end of a standard **singly linked list** is **O(n)** (linear time), where n is the number of nodes in the list.

Discuss : Yes, you typically need to traverse the entire list.

Arrays are memory-efficient in terms of overhead because they only store the data values.

Linked Lists require extra memory for the **pointers** (or references) in each node.

```
void InsertEnd(int value) {
    Node* newNode = new Node{value, nullptr};
    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
    n++;
}
```

Challenge 3 – insert in the middle

Linked List: Slow to find the location, but **fast to link** the nodes.

Array: Fast to find the location, but **slow to move** the data.

```
void InsertMid(int value, int pos ){

    if (pos > n)
    {
        cout << "out of node range \n" << endl;
        return ;
    }
    if (pos == 0)
    {
        InsertFront(value);
        return ;
    }
    if (pos == n )
    {
        InsertEnd(value);
        return ;
    }
    cur = head;
    for (int i = 0; i < pos - 1; i++)
    {
        cur = cur ->next;
    }
    Node* newNode = new Node{value, nullptr};
        newNode->next = cur ->next;
        cur -> next = newNode;
        n++;
```

challenge 4

The head pointer must be updated to point to the second node in the list.

The original head node (the one being deleted) is temporarily held by a pointer so its memory can be freed, and then the main head pointer is set to the current head.next.

```
}
void deleteFront(){
    if (n == 0)
    {
        cout << "No node to delete"<< endl;
        return;
    }
    Node * Cur = head;
    head = head-> next;
    delete cur ;
    n--;
```

Challenge 5

the traversal until curr.next is null. At this point, curr is the last node (the one to be deleted), and **prev is the second-to-last node** (the one need to modify).

```
}
void deleteEnd(){
    if( n == 0){
        cout << "No node to delete" << endl;
        return ;
    }
    if (head ->next == 0)
    {
        delete head;
        head = 0;
        return;
    }
    Node* cur = head;
    while (cur->next->next != nullptr)
    {
        cur = cur ->next;
    }
    delete cur -> next;
    cur -> next = nullptr;
```

Challenge 6

Complexity: O(n) (Linear Time, due to traversal to find the spot).

Arrow Change: Only **one** pointer changes: the **next pointer of the preceding node** is updated to point to the deleted node's successor.

Memory Leak: Forgetting to free the deleted node's memory creates a **memory leak**, where the occupied space remains reserved and inaccessible, leading to resource exhaustion over time.

```
void deleteMid(int pos){

    if (head == nullptr)
    {
        cout << "List is empty" << endl;
        return ;
    }
    if (pos <= 0)
    {
        cout << "Invalid position";
        return;
    }
    Node * cur = head ;
    for (int i = 0 ; cur != nullptr && i < pos -1 ;
    {
        cur = cur ->next;
    } Node* Nodedelete = cur->next;
    cur->next = Nodedelete->next;
    delete Nodedelete;
}
```

Challenge 7

Linked List Traversal: You must start at the head and follow the next **pointer** sequentially from node to node. You cannot skip or jump ahead. This is **sequential access**.

Array arr[i] Access: Elements are stored in contiguous memory. The computer calculates the memory address of the element directly using the base address and the element size. This is **random access** and is O(1).

```
void Display(){
        Node*cur = head ;
        while (cur != nullptr)
        {
            cout << cur ->value << "->";
            cur = cur ->next;
        }
        cout << "Null"<< endl;
    }
};
```

Challenge 8

The complexity remains O(n) overall because you first have to **traverse** the list to find the two nodes and their preceding nodes.

```
}
void swapNodes(int pos1, int pos2) {
    if (pos1 == pos2) {
        cout << "Both positions are the same — nothi
        return;
    }

    if (pos1 <= 0 || pos2 <= 0 || pos1 > n || pos2 >
        cout << "Invalid positions." << endl;
        return;
    }

    // Ensure pos1 < pos2 for simplicity
    if (pos1 > pos2) swap(pos1, pos2);

    Node* prev1 = nullptr;
    Node* curr1 = head;
    for (int i = 1; i < pos1; i++) {
        prev1 = curr1;
        curr1 = curr1->next;
    }

    Node* prev2 = nullptr;
    Node* curr2 = head;
    for (int i = 1; i < pos2; i++) {
        prev2 = curr2;
        curr2 = curr2->next;
    }
    // If any node is missing (invalid positions)
    if (curr1 == nullptr || curr2 == nullptr) {
        cout << "Position out of range." << endl;
        return;
    }

```

Challenge9

Random access means getting the element at index.

Arrays can compute this address instantly. Linked lists must traverse five nodes to reach the sixth element, making it an O(n) operation.

```
void searchAll(int value) {
    if (head == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }

    Node* cur = head;
    int pos = 1;
    bool found = false;

    while (cur != nullptr) {
        if (cur->value == value) {
            cout << "Value " << value << " found at
            found = true;
        }
        cur = cur->next;
        pos++;
    }

    if (!found) {
        cout << "Value " << value << " not found in
    }
}
```

Challenge 10

Implementing Stacks and Queues: It provides an essential O(1) time complexity for operations like pushing/popping or enqueuing/dequeuing, which arrays cannot match at the front.

Dynamic Memory: It adapts to size changes instantly and cheaply (O(1) node creation) without the costly, occasional O(n) array resizing/copying penalty.

1 **O(1) LL but O(n) Array: Insert/Delete at the Beginning.**

2 **Clearly Faster in Arrays: Access by Index (Random Access)**, which is O(1) in arrays.

3 **Why Memory Management?** To **prevent memory leaks** in languages without automatic garbage collection, as deleted nodes' memory must be explicitly freed.

4 **head Pointer:** Represents the **starting point** of the list; it holds the address of the first node.

5 **Losing head Pointer:** The **entire list is lost** and becomes unreachable, effectively creating a **memory leak** for all the nodes.