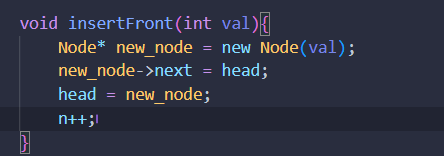
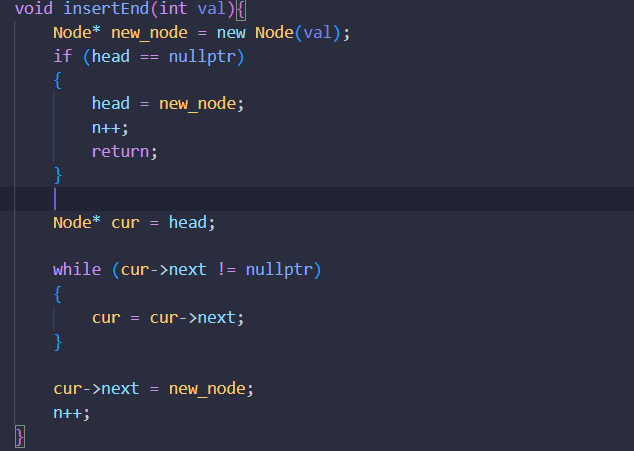
## Challenge 1 — Insert at the Front

Code:  


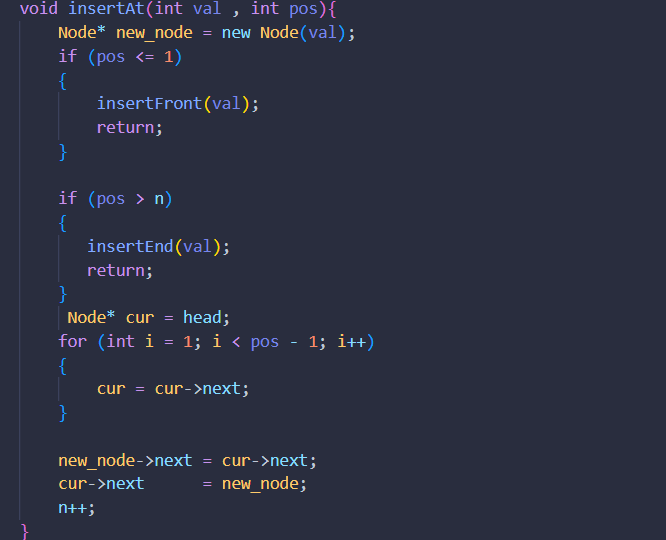
Explanation:  
Creates a new node and sets its next pointer to the current head, then updates head to point to the new node. This operation takes O(1) time since only two pointer changes occur.  
In arrays, inserting at index 0 requires shifting all elements to the right (O(n)).

## Challenge 2 — Insert at the End

Code:  


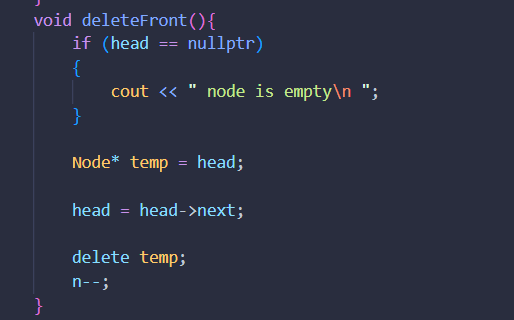
Explanation:  
Traverses to the last node (whose next is null) and adds a new node after it. Takes O(n) time since traversal is required. Using a tail pointer could make it O(1). Arrays append elements in O(1) if space is available.

## Challenge 3 — Insert in the Middle

Code:  


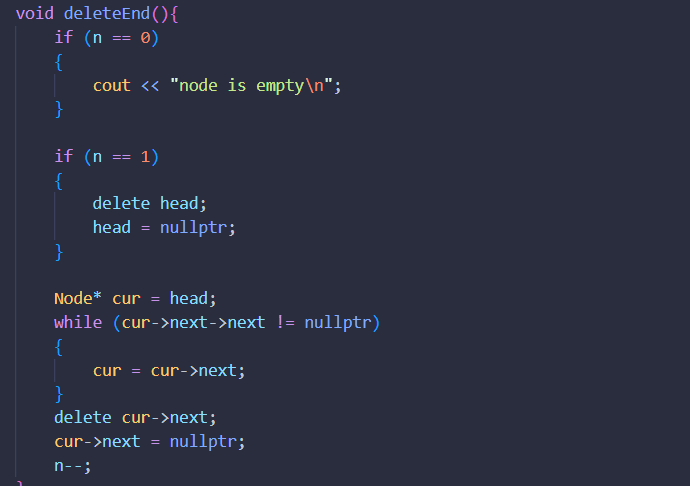
Explanation:  
Finds the position, then updates two pointers: (1) new\_node->next = cur->next and (2) cur->next = new\_node. Complexity O(n). Arrays require shifting elements, also O(n), but involve more memory movement.

## Challenge 4 — Delete from the Front

Code:  


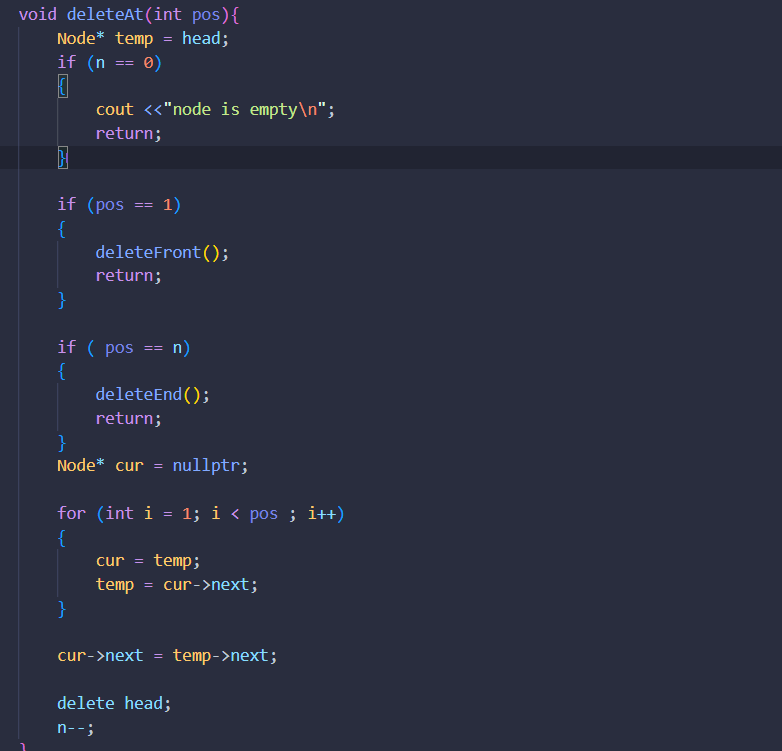
Explanation:  
Moves head to the second node and deletes the first. Takes O(1) time. Arrays need to shift all remaining elements, O(n).

## Challenge 5 — Delete from the End

Code:  


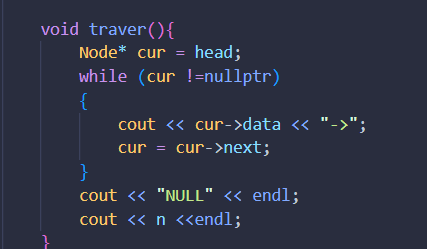
Explanation:  
Traverses to the second-to-last node, deletes the last, and sets next to null. Takes O(n) time since traversal is needed. Arrays can delete the last element instantly, O(1).

## Challenge 6 — Delete from the Middle

Code:  


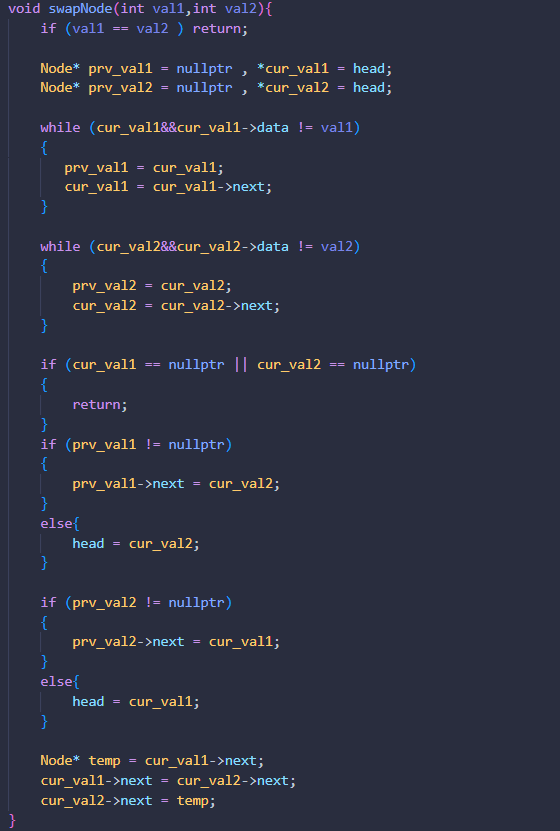
Explanation:  
Finds the node before the target, changes its next pointer to skip the deleted node, and frees memory. Forgetting to delete causes a memory leak. Complexity O(n). Arrays require element shifting.

## Challenge 7 — Traverse the List

Code:  


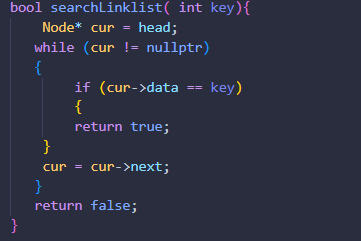
Explanation:  
Traverses node by node until NULL, printing each element. Complexity O(n). In arrays, we can directly access any index in O(1), unlike sequential traversal in linked lists.

## Challenge 8 — Swap Two Nodes

Code:  


Explanation:  
Swaps two nodes by adjusting their links, not data. Takes O(n) to locate nodes. Swapping data is simpler but doesn’t work for complex structures.

## Challenge 9 — Search in Linked List

Code:  


Explanation:  
Sequentially checks each node’s data until it matches the key or reaches the end. This is similar to linear search in arrays, O(n). Linked lists do not support random access.

## Challenge 10 — Compare with Arrays

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Array Complexity | Linked List Complexity | Explanation |
| Access element | O(1) | O(n) | Arrays have index-based access; lists require traversal. |
| Insert at front | O(n) | O(1) | Arrays shift elements; lists just move head. |
| Insert at end | O(1)\* | O(n) | Lists need traversal unless tail pointer is used. |
| Insert in middle | O(n) | O(n) | Both must find the position. |
| Delete from front | O(n) | O(1) | Arrays shift; lists move head. |
| Delete from end | O(1) | O(n) | Lists need to find the previous node. |
| Delete from middle | O(n) | O(n) | Both need traversal. |
| Search | O(n) | O(n) | Both linear search. |

\*O(1) for arrays only if resizing is not required.

## Summary

The code in insert\_front.cpp implements and demonstrates all key linked list operations: insertion, deletion, traversal, swapping, and searching. Linked lists provide efficient insertions and deletions but lack random access, making them suitable for dynamic data where element movement is frequent.