1. Write Algorithm/Pseudocode for following operations on a Singly Linked Lists

a) Insert at Beginning

Algorithm InsertAtBeginning(head, data):

newNode ← new Node

newNode.data ← data

newNode.next ← head

head ← newNode

return head

b) Insert at End

Algorithm InsertAtEnd(head, data):

newNode ← new Node

newNode.data ← data

newNode.next ← NULL

if head = NULL:

head ← newNode

return head

temp ← head

while temp.next ≠ NULL:

temp ← temp.next

temp.next ← newNode

return head

c) Insert After a Given Node

Algorithm InsertAfterNode(head, key, data):

temp ← head

while temp ≠ NULL and temp.data ≠ key:

temp ← temp.next

if temp = NULL:

Print "Key not found"

return head

newNode ← new Node

newNode.data ← data

newNode.next ← temp.next

temp.next ← newNode

return head

d) Insert Before a Given Node

Algorithm InsertBeforeNode(head, key, data):

if head = NULL:

return NULL

if head.data = key:

return InsertAtBeginning(head, data)

prev ← NULL

curr ← head

while curr ≠ NULL and curr.data ≠ key:

prev ← curr

curr ← curr.next

if curr = NULL:

Print "Key not found"

return head

newNode ← new Node

newNode.data ← data

newNode.next ← curr

prev.next ← newNode

return head

e) Delete from Beginning

Algorithm DeleteFromBeginning(head):

if head = NULL:

Print "List is empty"

return NULL

temp ← head

head ← head.next

Free temp

return head

f) Delete from End

Algorithm DeleteFromEnd(head):

if head = NULL:

Print "List is empty"

return NULL

if head.next = NULL:

Free head

return NULL

prev ← NULL

curr ← head

while curr.next ≠ NULL:

prev ← curr

curr ← curr.next

prev.next ← NULL

Free curr

return head

g) Delete After a Given Node

Algorithm DeleteAfterNode(head, key):

temp ← head

while temp ≠ NULL and temp.data ≠ key:

temp ← temp.next

if temp = NULL or temp.next = NULL:

Print "Deletion not possible"

return head

nodeToDelete ← temp.next

temp.next ← nodeToDelete.next

Free nodeToDelete

return head

h) Display (Traversal

Algorithm DisplayList(head):

temp ← head

while temp ≠ NULL:

Print temp.data

temp ← temp.next

i) Search for an Element

Algorithm SearchElement(head, key):

temp ← head

position ← 1

while temp ≠ NULL:

if temp.data = key:

Print "Found at position", position

return True

temp ← temp.next

position ← position + 1

Print "Not Found"

return False

j) Count Nodes

Algorithm CountNodes(head):

count ← 0

temp ← head

while temp ≠ NULL:

count ← count + 1

temp ← temp.next

return count

k) Reverse Linked List

Algorithm ReverseList(head):

prev ← NULL

curr ← head

next ← NULL

while curr ≠ NULL:

next ← curr.next

curr.next ← prev

prev ← curr

curr ← next

head ← prev

return head

2. Apriori Time and Space Complexity Analysis of Singly Linked List Operations

a) Insert at Beginning

Time Complexity:  
O(1) — Insert at beginning involves just creating a new node and updating the head pointer, which is a constant time operation.

Space Complexity:  
O(1) — Only one new node is created regardless of list size.

b) Insert at End

Time Complexity:  
O(n) — Need to traverse the entire list to reach the last node (unless tail pointer maintained).

Space Complexity:  
O(1) — Only one new node allocated.

c) Insert After a Given Node

Time Complexity:  
O(n) — Must search for the node with the given key by traversing the list; once found, insertion is O(1).

Space Complexity:  
O(1) — Only one new node allocated.

d) Insert Before a Given Node

Time Complexity:  
O(n) — Need to traverse the list to find the node and keep track of previous node.

Space Complexity:  
O(1) — Only one new node allocated.

e) Delete from Beginning

Time Complexity:  
O(1) — Deletion is just changing the head pointer and freeing the node.

Space Complexity:  
O(1) — No additional space used.

f) Delete from End

Time Complexity:  
O(n) — Need to traverse the list to find the second last node to update its next pointer.

Space Complexity:  
O(1) — No extra space allocated.

g) Delete After a Given Node

Time Complexity:  
O(n) — Need to find the node with the key, then delete the next node (if exists).

Space Complexity:  
O(1) — No extra space allocated.

h) Display the Entire List (Traversal)

Time Complexity:  
O(n) — Must visit each node exactly once.

Space Complexity:  
O(1) — Just uses temporary pointer variables.

i) Search for an Element

Time Complexity:  
O(n) — May need to check every node.

Space Complexity:  
O(1) — Only pointer and counter variables used.

j) Count Number of Nodes

Time Complexity:  
O(n) — Must traverse all nodes to count.

Space Complexity:  
O(1) — Only a counter variable is needed.

k) Reverse the Singly Linked List

Time Complexity:  
O(n) — Must traverse all nodes once, changing the next pointers.

Space Complexity:  
O(1) — Uses a few pointers for traversal and reversal; no extra nodes allocated.

|  |  |  |
| --- | --- | --- |
| Operation | Time Complexity | Space Complexity |
| Insert at End | O(n) | O(1) |
| Insert After Node | O(n) | O(1) |
| Insert Before Node | O(n) | O(1) |
| Delete from Beginning | O(1) | O(1) |
| Delete Insert at Beginning | O(1) | O(1) |
| from End | O(n) | O(1) |
| Delete After Node | O(n) | O(1) |
| Display List | O(n) | O(1) |
| Search Element | O(n) | O(1) |
| Count Nodes | O(n) | O(1) |
| Reverse List | O(n) | O(1) |

3. How can you optimise the insertion of an element at the end. Discuss.

void insertAtEndOptimized(Node \*\*head, Node \*\*tail, int data) {

Node \*newNode = (Node\*) malloc(sizeof(Node)); // Allocate memory for the new node

newNode->data = data;

newNode->next = NULL; // Set the next pointer of the new node to NULL

if (\*head == NULL) { // If the list is empty

\*head = \*tail = newNode; // Both head and tail point to the new node

} else {

(\*tail)->next = newNode; // Link the current tail to the new node

\*tail = newNode; // Update the tail to point to the new node

}

}

4. Discuss about why it’s not possible to delete a node before a given node

Why can't we delete the node before a given node directly?

Because:

We can't go backwards:

A node doesn’t have a pointer to its previous node.

So, given a pointer to a node, there’s no way to find the node that comes before it without starting from the head of the list and traversing.

To delete a node, we need access to its previous node:

To delete a node X, we need to change the .next pointer of the node before X so it skips over X.

If we want to delete the node before a given node, we would need access to the node two steps back to adjust the pointer correctly.

The given node has no knowledge of its predecessor:

A singly linked list doesn't support backward traversal unless we explicitly maintain previous pointers — which would make it a doubly linked list.

It’s only possible if:

You're given the head of the list (so you can traverse it),

And the target node (the node before which deletion is to happen) is not the first or second node (since there’s no node before the head or a second node if the head is to be deleted).

In that case, you can:

Traverse the list starting from the head,

Keep track of the previous and current node,

When you find the node that comes right after the node you want to delete, adjust pointers.

11. Discuss about the Time and Space complexity of various operations on Singly Linked Lists with your friends.

* Time and Space Complexity of Singly Linked List Operations

Singly linked lists are fundamental data structures that consist of nodes where each node contains data and a pointer to the next node. Understanding the efficiency of various operations on singly linked lists is important to write optimal code.

**Time Complexity**

**Insertion at Beginning:**

Inserting a new node at the start is very efficient. It requires updating the head pointer to point to the new node and the new node’s next pointer to the old head. This operation always takes constant time, O(1).

**Insertion at End:**

If the list maintains a tail pointer, insertion at the end is also O(1) because the new node can be directly linked to the tail. Without a tail pointer, the entire list must be traversed to find the last node, resulting in O(n) time complexity.

**Insertion at Given Position:**

Inserting at a particular position generally takes O(n) time in the average and worst cases because you must traverse the list to reach the insertion point. It is O(1) in the best case if the position is at the beginning.

**Deletion from Beginning:**

Deleting the first node is straightforward; it just involves moving the head pointer to the next node and freeing the old head. This operation is O(1).

**Deletion from End:**

Removing the last node requires traversing the list to find the second-last node so that its next pointer can be set to NULL. This operation takes O(n) time.

**Deletion by Key (Search + Delete):**

Searching for a node by its value or key requires checking nodes one by one, so it takes O(n) time in the average and worst cases. If the node is found at the head, the deletion is O(1).

**Search:**

Searching for an element is a sequential process, requiring traversal from the head node until the element is found or the list ends. It has O(n) average and worst-case complexity.

**Traversal / Display All:**

Displaying or traversing all nodes naturally requires visiting each node once, so the time complexity is O(n).

**Counting Nodes:**

Counting the total number of nodes requires traversal, hence O(n), unless a global counter is maintained during insertions and deletions, which allows O(1) counting.

**Space Complexity**

Operations such as insertion and deletion require only a constant amount of extra memory, O(1), mostly for pointer manipulation or node creation.

Searching and traversal also use O(1) extra space since they only maintain a pointer for iteration.

The total space required for storing the entire list is proportional to the number of nodes, i.e., O(n), where each node stores data and a pointer.