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) |