SINGLY LINKED LIST OPERATIONS ARE:

- Traversal – Access the nodes of the list.
- Insertion – Adds a new node to an existing linked list.
- Deletion – Removes a node from an existing linked list.
- Search – Finds a particular element in the linked list.

Algorithm: Traverse

Step 1: [INITIALIZE] SET PTR = HEAD
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3: Apply process to PTR -> DATA
Step 4: SET PTR = PTR->NEXT
[END OF LOOP]
Step 5: EXIT

Algorithm: InsertAtBeginning

Step 1: IF AVAIL = NULL
Write OVERFLOW
Go to Step 7
[END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = HEAD
Step 6: SET HEAD = NEW_NODE
Step 7: EXIT

Algorithm: InsertAtEnd

Step 1: IF AVAIL = NULL
Write OVERFLOW
Go to Step 10
[END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = HEAD
Step 7: Repeat Step 8 while PTR -> NEXT != NULL

Step 8: SET PTR = PTR -> NEXT
[END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT

Algorithm: InsertAfterAnElement

Step 1: IF AVAIL = NULL
Write OVERFLOW
Go to Step 12
[END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = HEAD
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PREPTR -> DATA != NUM
Step 8: SET PREPTR = PTR
Step 9: SET PTR = PTR -> NEXT
[END OF LOOP]
Step 1 : PREPTR -> NEXT = NEW_NODE
Step 11: SET NEW_NODE -> NEXT = PTR
Step 12: EXIT

Algorithm: DeleteFromBeginning

Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 5
[END OF IF]
Step 2: SET PTR = HEAD
Step 3: SET HEAD = HEAD -> NEXT
Step 4: FREE PTR
Step 5: EXIT

Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 8
[END OF IF]
Step 2: SET PTR = HEAD
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != NULL

Step 4: SET PREPTR = PTR

Step 5: SET PTR = PTR -> NEXT
[END OF LOOP]
Step 6: SET PREPTR -> NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT

Algorithm: DeleteAfterANode

Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 10
[END OF IF]
Step 2: SET PTR = HEAD
Step 3: SET PREPTR = PTR
Step 4: Repeat Steps 5 and 6 while PREPTR -> DATA != NUM
Step 5: SET PREPTR = PTR
Step 6: SET PTR = PTR -> NEXT
[END OF LOOP]
Step 7: SET TEMP = PTR

Step 8: SET PREPTR -> NEXT = PTR -> NEXT
Step 9: FREE TEMP
Step 10 : EXIT

Algorithm: Search

Step 1: [INITIALIZE] SET PTR = HEAD

Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3: If ITEM = PTR -> DATA
SET POS = PTR

Go To Step 5
ELSE
SET PTR = PTR -> NEXT
[END OF IF]
[END OF LOOP]

Step 4: SET POS = NULL
Step 5: EXIT

DOUBLY LINKED LIST OPERATIONS ARE:

- Traversal – Access the nodes of the list.
- Insertion – Adds a new node to an existing linked list.
- Deletion – Removes a node from an existing linked list.
- Search – Finds a particular element in the linked list.

Algorithm for traverse

1. Set PTR = NEXT[START]
2. Repeat steps 3 & 4 while PTR != START
3. Apply the PROCESS to DATA[PTR]
4. Set PREV[PTR] = PTR and PTR = NEXT[PTR]
5. Exit.

Algorithm for traverse

1. [OVERFLOW?] If AVAIL = NULL, then Write: OVERFLOW & Exit.
2. Set NEW = AVAIL, AVAIL= NEXT[AVAIL] and DATA[NEW] = ITEM.
3. Set NEXT[LOCA] = NEW and NEXT[NEW] =LOCB, PREV[LOCB]= NEW and PREV[NEW] =LOCA.
4. Exit.

Algorithm for deletion

1. Set NEXT[PREV[LOC]] = NEXT[LOC] and PREV[NEXT[LOC]]=PRV[LOC]
2. Set NEXT[LOC] = AVAIL and AVAIL = LOC
3. Exit

Algorithm for search

1. SetPTR=NEXT[START].
2. Repeat while DATA[PTR] = ITEM & PTR!=START:
   - Set PREV[PTR]=PTR and PTR= NEXT[PTR]
3. If DATA[PTR] = ITEM, then:
   - Set LOC = PTR
   - Else LOC = NULL
4. Exit.

CIRCULAR LINKED LIST OPERATIONS ARE:

insert − Inserts an element at the start of the list.
delete − Deletes an element from the start of the list.
display − Displays the list.

Algorithm for insertion at the beginning

1] Create a newNode with given value.
2] Check whether list is Empty (head == NULL)
3] If it is Empty then, set head = newNode and
newNode→next = head .
4] If it is Not Empty then, define a Node pointer 'temp'
and initialize with 'head'.

5] Keep moving the 'temp' to its next node until it reaches to the last node (until 'temp → next == head').
6] Set 'newNode → next =head', 'head = newNode' and 'temp → next = head'.

Algorithm for insertion at the end

1]Create a newNode with given value.
2]Check whether list is Empty (head == NULL).
3]If it is Empty then, set head = newNode and newNode → next = head.
4]If it is Not Empty then, define a node pointer temp and initialize with head.
5]Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next == head).
6]Set temp → next = newNode and newNode → next = head.

Algorithm for inserting after a specific node

1]Create a newNode with given value.
2] Check whether list is Empty (head == NULL)
3] If it is Empty then, set head = newNode and newNode → next = head.
4] If it is Not Empty then, define a node pointer temp and initialize with head.
5] Keep moving the temp to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).
6] Every time check whether temp is reached to the last node or not. If it is reached to last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp to next node.
7] If temp is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).
8] If temp is last node then set temp → next = newNode and newNode → next = head.
9] If temp is not last node then set newNode → next = temp → next and temp → next = newNode.

Algorithm for deletion at beginning

1]Check whether list is Empty (head == NULL)
2]If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
3] If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize both 'temp1' and 'temp2' with head.
4] Check whether list is having only one node (temp1 →

next == head)
5] If it is TRUE then set head = NULL and delete temp1 (Setting Empty list conditions)
6] If it is FALSE move the temp1 until it reaches to the last node. (until temp1 → next == head)
7]Then set head = temp2 → next, temp1 → next = head and delete temp2.

Algorithm for deletion at end

1]Check whether list is Empty (head == NULL)
2]If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
3] If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.
4]Check whether list has only one Node (temp1 → next == head)
5]If it is TRUE. Then, set head = NULL and delete temp1. And terminate from the function. (Setting Empty list condition)
6]If it is FALSE. Then, set 'temp2 = temp1 ' and move temp1 to its next node. Repeat the same until temp1 reaches to the last node in the list. (until temp1 → next == head)
7]Set temp2 → next = head and delete temp1.

Algorithm for deletion at middle

1]Check whether list is Empty (head == NULL)
2]If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
3]If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.
4]Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set 'temp2 = temp1' before moving the 'temp1' to its next node.
5]If it is reached to the last node then display 'Given node not found in the list! Deletion not possible!!!'. And terminate the function.
6]If it is reached to the exact node which we want to delete, then check whether list is having only one node (temp1 → next == head)
7] If list has only one node and that is the node to be deleted then set head = NULL and delete temp1 (free(temp1)).
8]If list contains multiple nodes then check whether temp1 is the first node in the list (temp1 == head).
9]If temp1 is the first node then set temp2 = head and keep moving temp2 to its next node until temp2 reaches to the last node. Then set head = head → next, temp2 → next = head and delete temp1.
10]If temp1 is not first node then check whether it is last node in the list (temp1 → next == head).

11]If temp1 is last node then set temp2 → next = head and delete temp1 (free(temp1)).
12]If temp1 is not first node and not last node then set temp2 → next = temp1 → next and delete temp1 (free(temp1)).

Algorithm for Displaying

1] Check whether list is Empty (head == NULL)
2] If it is Empty, then display 'List is Empty!!!' and terminate the function.
3] If it is Not Empty then, define a Node pointer 'temp' and initialize with head.
4] Keep displaying temp → data with an arrow (--->) until temp reaches to the last node
5] Finally display temp → data with arrow pointing to head → data.

Algorithm for Reversing

1]Initialize three pointer variables, last = head, cur = head->next and prev = head. Where head is pointer to first node in the circular linked list.
2]Move head node ahead i.e. head = head->next;
3]Link current node with previous node i.e. cur->next = prev;
4]Make previous node as current node i.e. prev = cur;
5]Make current node as head node i.e. cur = head
6]Repeat step 2 to 5 till head != last
7]After linking all nodes in loop, link current node with previous node i.e. cur- >next = prev;
8]Finally move head node at its position. Move head node to last i.e. head = prev

Algorithm for count nodes in circular linked list

1]Create a Circular Linked List and assign reference of first node to head.
2]Initialize count = 0; variable to store total nodes in list.
3]Initialize another variable to traverse list, say current = head;.
4]Increment count++ and current = current->next;.
5]Repeat step 4 till you reach head node after traversing once.