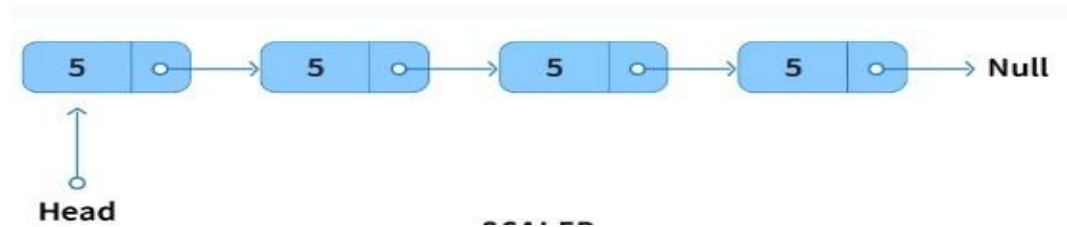


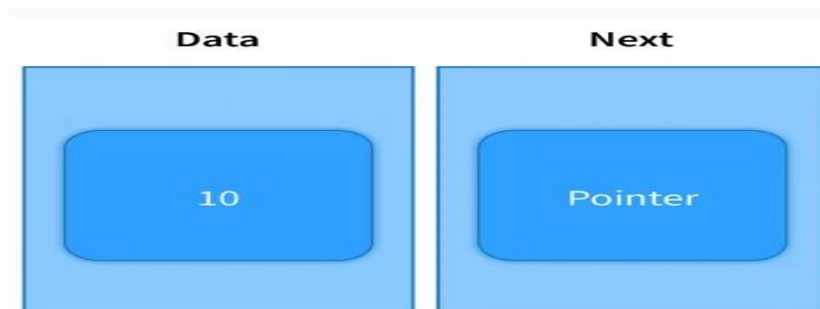
LINKED-LIST

A linked-list is a grouping of elements where each entry points to the one that comes after it in the sequence. A node is any one of the Linked List's elements. Multiple links are connected to form a Linked List.



A node is composed of two components:

1. Data: This is the real, unprocessed information you wish to keep up to date. It could be a character, a number, a video item, etc.
2. Pointer to another node: It keeps track of the neighbouring node's link.

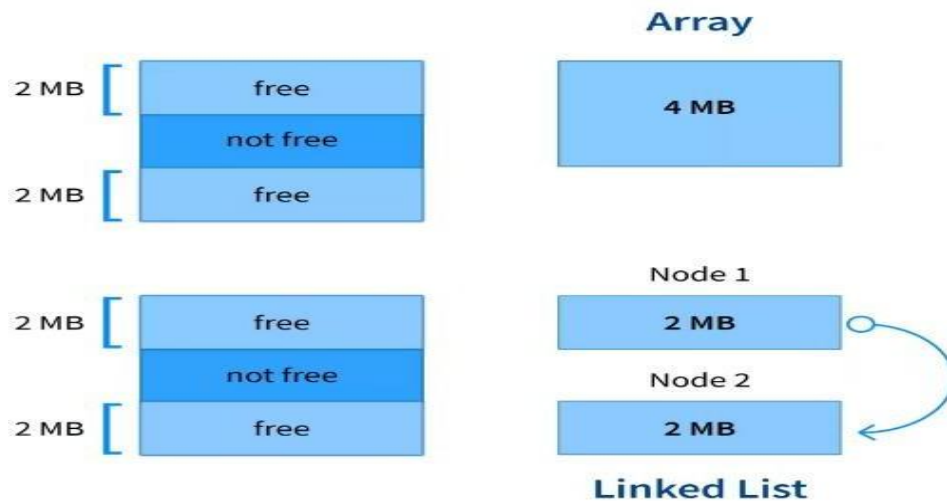


linked list over an array:

Dynamic Dimensions Arrays are limited in size. An array cannot be changed after its size is specified. On the other hand, the linked list's size is modifiable during runtime. Memory Consumption: Memory must be allocated to arrays in a continuous fashion. Memory for linked lists, however, can be dynamically allocated. A linked list's nodes are all connected by pointers and kept in non- contiguous memory regions.

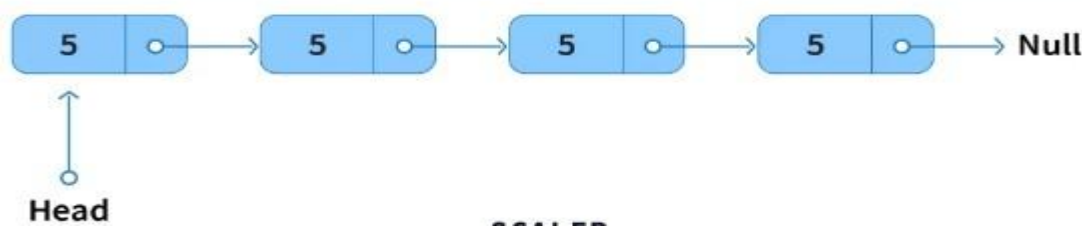
For example: consider system has 4 MB of free space. Yet, the open area is not continuous. Currently, 4MB of linked lists can be stored here with ease, but 4MB of arrays cannot. Therefore, linked lists make the most of memory.

Quick Insertion and Removal of array, the period of time complexness of deletion and inserting is $O(n)$. On the other hand, $O(1)$ can be used for deletion and insertion in linked lists.



A single linked list: what is it?

A singly linked list is a particular example of a generic linked list. If we start at the first node in the list, we can only go in one way because every node in a singly linked list links to only the next node in the sequence..



The alone subsequent element's address is kept apart from the contents in the single linked list's node. This is how a node appears in its C representation

```
" Node{
  int data;
  Node next;
}"
```

A unique pointer called "head" for each linked list . This pointer contains the memory address of the initial node in the list. The following element can no thirster be present on the last node. Therefore, we designate NULL to the next element in the linked list to denote its termination.

1. Inserting in a single linked list

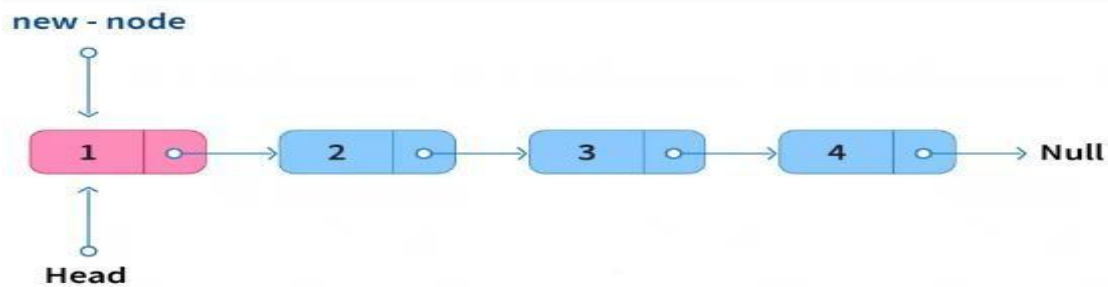
The following methods can be used to insert data into a single linked list:

Insertion at the outset To add a new node to the start of a singly linked list, perform these steps:.

Point the newly created node at HEAD.

Set the new node's HEAD to refer there.

O(1) to insert a new node at the beginning.



```

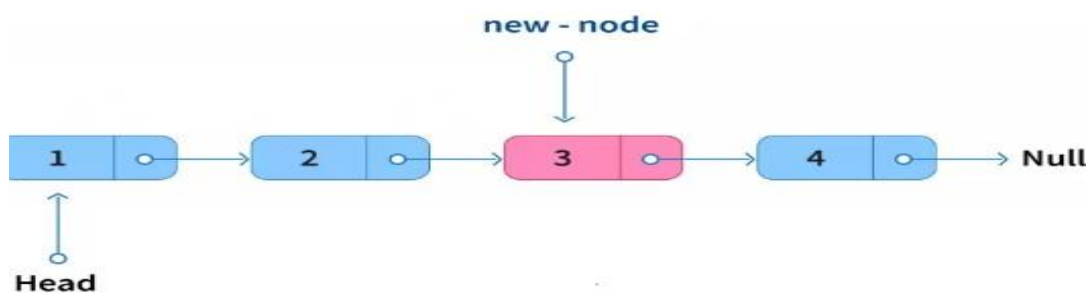
void insertAtStart(Node newNode, Node head){
    newNode.data = 10;
    newNode.next = head;
    head.next = newNode;
}

```

- **Insertion following a few Nodes** The process of adding a new node to a singly linked list after an existing node is as follows:

Insert the new node once you've reached the intended node.

- o Configure the new node to point to the element that comes after the existing node.
- o Aim the new node directly at the current node. Adding a new node after an existing one is an $O(N)$ operation.



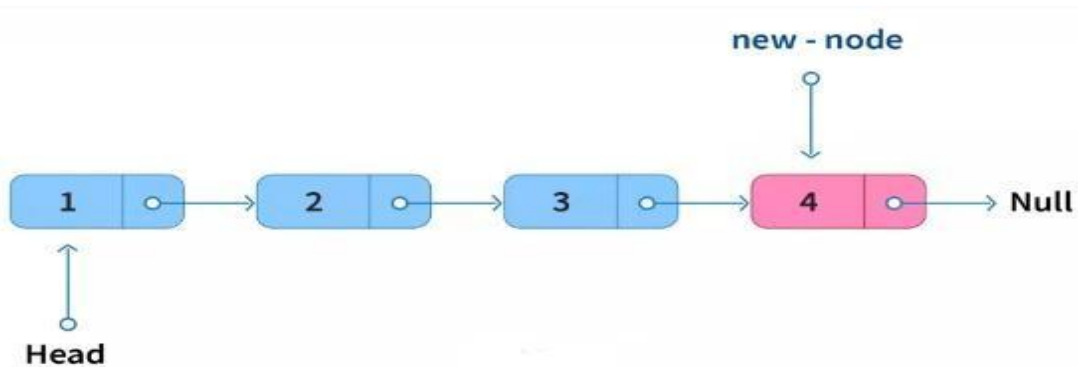
```

void insertAfterTargetNode(Node newNode, Node head, int target){
    newNode.data = 10;
    Node temp = head;
    while(temp.data != target){
        temp = temp.next;
    }
    newNode.next = temp.next;
    temp.next = newNode;
}

```

- **Insertion at the end** Insertion of a new node at the end of a singly linked list is performed in the following way,

- o Go from beginning to end of the list and finish at the final node.
- o Point the new node at the end of the node. In order to indicate the end of the list, set the new node to point to null. It takes $O(N)$ operations to insert a new node at the end.



```

“void insertAtEnd(Node newNode, Node head){
  newNode.data = 10;
  Node temp = head;
  while(temp.next != null){
    temp = temp.next;
  }
  temp.next = newNode;
  newNode.next = null;
}”

```

2. Deletion

Deletion at the start

1. singly linked list's initial node in the following way:
2. Point the HEAD to the following element.
3. In a singly linked list, deleting the initial node is an $O(1)$ operation.

```

“void deleteAtFirst(Node head){
  head = head.next;
}”

```

Deletion at the middle

The deletion activity after a particular node can be operated in the following way,

- a. While at the particular node, remove the node from existence.
- b. Set the current node to point to the element after this one.
- c. An $O(N)$ operation is when a node is deletion after a certain node.

```

“void deleteAfterTarget(Node head, int target){
  Node temp = head;
  while(temp.data != target){
    temp = temp.next;
  }
  temp.next = temp.next.next;
}”

```

Deletion at last

The process for deletion the last node is as follows:

- a. Locate the second-to-last node in the singly linked list.
- b. Set the second-to-last node point to zero.

Deleting the last node is an $O(N)$ operation.

```
"void deleteLast(Node head){  
    Node temp = head;  
    while(temp.next.next != null){  
        temp = temp.next;  
    }  
    temp.next = null;  
}"
```

3. Display

traverse the single linked list from first to last in order to show it in its entirety.

Linked list nodes cannot be arbitrarily accessed, in contrast to arrays. Therefore, we must go through all $(n-1)$ elements in order to get to the n -th element.

We acquire $O(N)$ time complexity in this operation since the complete linked list is visited. The full single linked list can be shown using the sample that follows.

```
"void display(Node head){  
    Node temp = head;  
    while(temp != null){  
        System.out.println(temp.data);  
        temp = temp.next;  
    }  
}"
```

4. Search

We must search the full linked list in order to find an entry in the singly linked list.

Every node undergoes a lookup to ascertain whether the target has been located; if so, the target node is returned; if not, we proceed to the subsequent element.

Searching an element in the singly linked list would cost us $O(N)$ operating time if, in the worst scenario, we end up visiting every node in the list.

```
"Node search(Node head, int target){  
    Node temp = head;  
    while(temp != null && temp.data != target){  
        temp = temp.next;  
    }  
    return temp;  
}"
```

Linked List Applications

Some of the linked list applications are as follows:

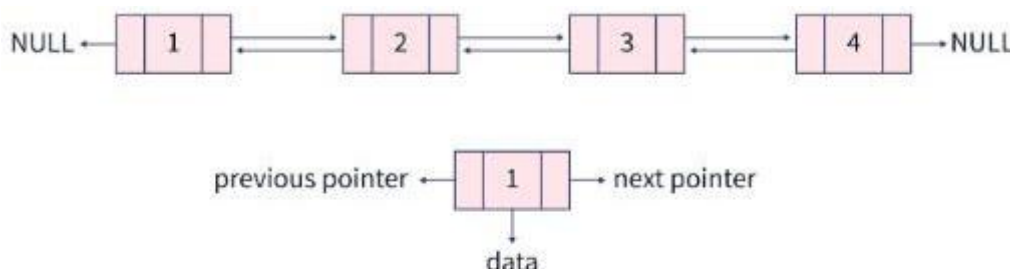
- To hold polynomials that are single or bivariate.
- The purpose serves as the base for certain data structures such as Graph, Stack, and Queue.
- File allocation strategies per operating system strategy.
- Monitor the secondary disk's spare space. It is possible to connect all of the empty areas.
- To determine which player will be played next in turn-based games, a circular linked list may be used. We move on to the next player when that person has finished their turn.
- To preserve data on things that may be accessed sequentially and linked to one another, such web pages, music, films, photos, and so forth.

Doubly Linked List

A doubly linked list, which permits traversal across the list in both forward and backward directions, is an alternative to a single linked list. Every node in the list has three pointers stored in it: one for the data itself, one for the node before it, and one for the node after it.

A Double Linked List in C:

A double linked list is an advanced data structures that is an improved version of a single linked list. An example of a linked data structures is a double linked list, which consists of links, or records, connected sequentially by pointers arranged in a chain. Each link in a doubly linked-list contains the data as well as pointers to the nodes that come before and after it in the sequence. The labels "previous" and "following" refer to these pointers, respectively. This pointer contains the memory locations of the previous and next points in the series. A unique node pointer head indicates the start of a double linked list. The start nodes previous pointer in a doubly linked list and the end node's next pointer points to either a sentinel value or a terminator. In C language, the sentinel value is the null pointer. Many activities can be accomplished with a doubly linked list, including showing the list from the head node down to the end and adding new nodes at specific positions or removing nodes from the list at specific positions.



The following elements must be defined in order to write the double linked list program in C:

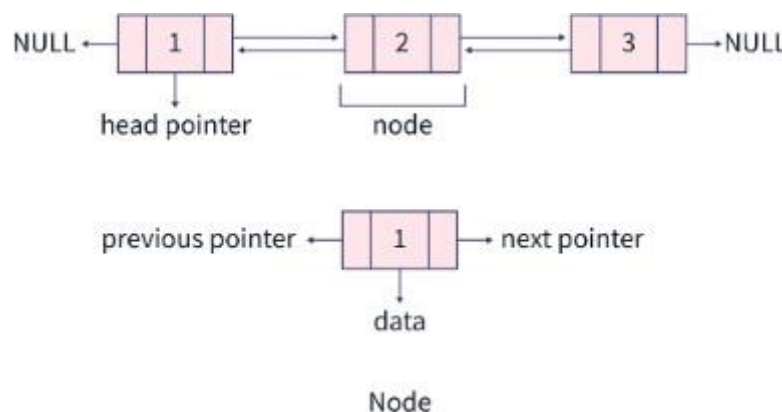
Node: The fundamental component of a double linked-list is the node. A double linked-list is created by connecting nodes with pointers. The data item, the next pointer, and the prior pointer are all stored on each node.

Next Pointer: In a double linked-list, it holds the address of the node that comes after in the sequence.

Previous Pointer: In a doubly linked list, it holds the memory address of the node that came before it in the sequence.

Data: It keeps track of the node's data value.

Head: This denotes where the double linked-list begins.



double-linked-list operations:

A doubly linked list can be used for numerous activities. Later in the post, we will go over each operation's implementation strategy and approach.

Traversing:

Traversing the linked list, checking at each node along the way from the head pointer to the finish. This typically occurs to accomplish a specific objective, such finding a node or showing all of the node's data, among other things.

Insertion at begin:

in this case, a new node comes before the double linked-list. The modified Head pointer now points to the newly inserted node.

Insertion at last:

In this case, a new node is added at the end of the double linked-list. There is no need to change the head pointer.

Insertion after a given node:

When receive a pointer to a node in the doubly linked list, we must place the new node after the node whose pointer we have been given. We can add more nodes at any time to a doubly linked list. A node can be inserted into a doubly linked node in four main ways: In this case, the goal is to insert the new node before the node to which the pointer is supplied.

Deletion

The process of removing a node from a doubly linked list while keeping the list's structure intact is called deletion. There are three circumstances in which a node can be eliminated from a double linked-list:

Deletion at beginning:

The double linked-list's starting node is eliminated. By changing the head pointer, the next node of the deleted node is highlighted. The removed node disappears from the memory.

Deletion of the node at a given position:

Removal of the node at a specific position: In this instance, the node at the designated location needs to be removed from the double linked-list.

Last deletion:

The last node of the doubly linked-list is removed.

Elimination of the node at a certain location: In this case, the double linked-list must have the node at the specified place deleted from it.

.

Searching :

The procedure of searching for a node involves comparing each node's data in a double linked-list with the designated item to be sought. The address or position of the node inside the linked list is frequently returned as the output. This address can be used to visit the same node if needed. If such a node does not exist, NULL is returned in the output.

Insert node at the front:

The new node in front of the doubly linked list is updated. This new node is now the head pointer.

Step 1: To construct a new node, use the data item. This node will be referred to as new_node.

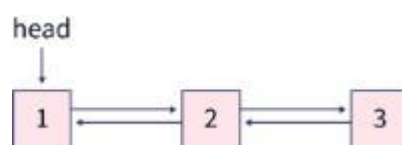
Step 2: Pass the next pointer of the new node to the head.

Step 3: If the head pointer is not NULL, assign the head's previous pointer to the new_node.

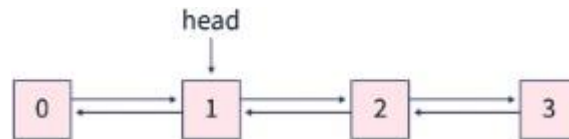
Step 4: Finally, update the head pointer.

Example:

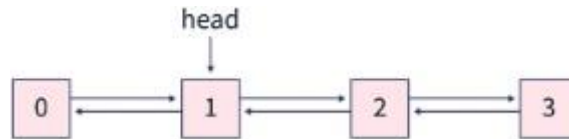
Think of a linked list with three nodes: $1 \leftrightarrow 2 \leftrightarrow 3$. The node with data 0 at the front is what we now wish to add.



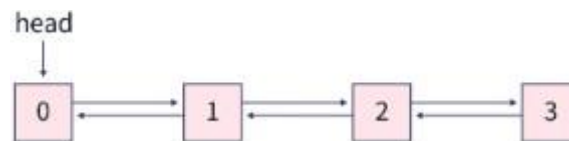
Initially, a new node is created with data 0, and its next pointer is assigned to the head pointer.



We now assign the head's previous pointer to the newly created node.



Lastly, we make updates to the head pointer.



note: In the function `insertAtFront`, we pass a Spanish pointer to the main head pointer. The head pointer will only be updated locally inside the function if we don't do this.

```

“void insertAtFront(struct Node** head, int data) {
    // Create a new struct node and allocate memory.
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    // Initialize data in the newly created node.
    newNode->data = data;
    // Make the next pointer of the new node point to the head.
    // In this way, this new node is added at front of the linked list.
    newNode->next = (*head);
    newNode->prev = NULL;
    // If the head is not NULL, then we update the prev pointer in the current head.
    // The prev pointer of the current head will point to the new node.
    if((*head) != NULL) {
        (*head)->prev = newNode;
    }
    // Update the head pointer to point to the new node.
    (*head) = newNode;
}”
  
```

Analysis of Complexity

Complexity of Time:

Since we are just changing a certain number of references when we add a new node at the front, the time complexity is $O(1)$.

Complexness of Time: $O(1)$

Complexness of Space -All we need to do is create a pointer to the new node in order to add it to the front. Since no more memory is required, the space complexity is $O(1)$.

Complexness of Space: $O(1)$

A C program that implements the double linked-list algorithm by inserting a node between two nodes

In this instance, we have to insert the new node after receiving a node.

Step 1: Using the provided data, create a fresh node called new_node.

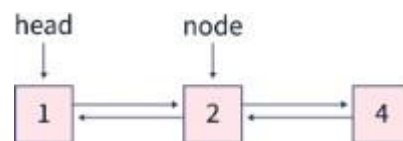
Assign new_node_next to node_next in step two.

Step 3: Assign new_node to node→next.

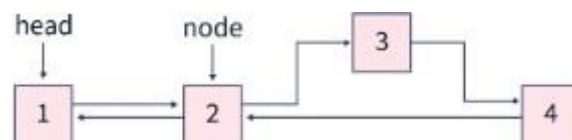
Step 4: Assign node to new_node → prev.

Step 5: Assign new_node → next → prev to new_node if new_node → next is not NULL.

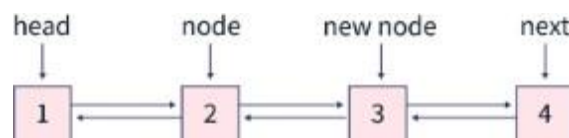
Suppose you have a linked list with three nodes: $1 \leftrightarrow 2 \leftrightarrow 4$. After the second node, we now wish to add the node with data 3. The reference to the second node is likewise provided to us.



The next pointer of the new node will be updated to the next pointer of the existing node first. Next, we assign the new node's pointer to the next pointer of the provided node.



The next pointer of the new node will be updated to the next pointer of the existing node first. Next, we assign the new node's pointer to the next pointer of the provided node.



```

“void insertAfterNode(struct Node* node, int data) {
    if(node == NULL) {
        printf("The given node cannot be NULL.");
        return ;
    }
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = node->next;
    node->next = newNode;
    newNode->prev = node;
    if(newNode->next != NULL) {
        newNode->next->prev = newNode;
    }
}”

```

C Program for the Initialization of the Doubly Linked List Algorithm's Node Removal

Step 1: set up a pointer curver to point to the h head pointer.

Step 2 is to increment the h head pointer if it is not NULL.

Step 3: Release the memory and set the curr pointer to NULL.

As Example.

Assume linked list that has three nodes: $1 \leftrightarrow 2 \leftrightarrow 3$. We now wish to remove the front node. In this instance, we just delete the first node in the series and update the head pointer to the next head pointer.

Finished Order: $2 \leftrightarrow 3$

```

“void deleteAtFront(struct Node** head) {
    // If the head pointer is NULL, then we cannot delete it.
    if((*head) == NULL) return ;
    struct Node* curr = *head;
    // Update the head pointer to the next node.
    *head = (*head)->next;
    // Make the previous head pointer NULL and free the memory.
    curr->next = NULL;
    free(curr);
}”

```

C Program to Remove a Code at the Double Linked List Algorithm's End

Step 1: set up a pointer curver to point to the h head pointer.

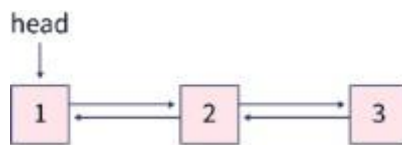
Step 2: Increase the curr pointer as long as $\text{curr} \rightarrow \text{next}$ is not NULL.

Step 3: Update $\text{curr} \rightarrow \text{prev} \rightarrow \text{next}$ and $\text{curr} \rightarrow \text{prev}$ as NULL in

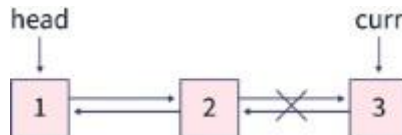
Step 4: Release the memory and set the curr pointer to NULL.

As example

assume a linked list that has three nodes: 1—2—3. At this point, we wish to remove the node.



First, we use a pointer curve to iterate to the final node. We now set curr→next to NULL.



```
"void deleteAtLast(struct Node** head) {  
    if((*head) == NULL) return ;  
    struct Node* curr = *head;  
    while(curr->next != NULL) {  
        curr = curr->next;  
    }  
    struct Node* prev = curr->prev;  
    prev->next = NULL;  
    curr->prev = NULL;  
    free(curr);  
}"
```

C Program for the Doubly Linked List Algorithm's Node Removal at a Specified Position

In this instance, the node at a given index position has to be deleted.

Step 1: The front node is deleted if position equals 1.

Step 2: Until position > 1, iterate a pointer cur

Step 3: The curr node must now be removed, and curr→prev and curr→next must be connected.

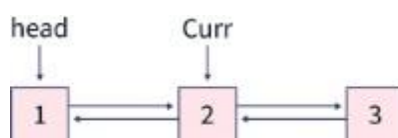
Assign curr→prev→next to curr→next & curr→next→prev to curr→prev in step four.

Step 5: Release the memory and set the cursor pointer to NULL.

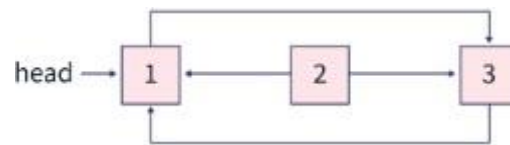
As Example:

Think of a linked list that has three nodes: 1↔2↔3. The second node in the list is what we now wish to remove.

First, we use a pointer curve to navigate to the second node.



Next, we establish a connection between the earlier present node and the subsequent node of the curr node.



```

void deleteAtPosition(struct Node** head, int position) {
    struct Node* curr = *head;
    if(position == 1) {
        deleteAtFront(head);
        return;
    }
    while(position > 1 && curr) {
        curr = curr->next;
        position--;
    }
    if(curr == NULL) {
        printf("Node at position is not present.\n");
        return;
    }
    struct Node* prevNode = curr->prev;
    struct Node* nextNode = curr->next;
    curr->prev->next = nextNode;
    if(nextNode != NULL) {
        nextNode->prev = prevNode;
    }
    curr->next = curr->prev = NULL;
    free(curr);
}
  
```

Circular-Linked-List

Introduction Circular-Linked-List

Circular-Linked-Lists are a kind of Linked List Data Structures, as the name implies. Before moving on to the Circular-Linked-List, over the Linked List ideas, if we choose to use arrays, our three items would be represented as {1, 2, 3}.

In contrast, every element in a linked list control a pointer to the element after it. would be represented as:



It has unique benefits and drawbacks in comparison to arrays. It is difficult to go around a linked list in a straightforward manner. "circle," mean that we may obtain the starting node from the last node after reached the last node in the list.

there are numerous real-world situations where it is necessary to repeatedly circle the elements on the list. Furthermore, as we've seen, the end node in a linked list leads to Null; as a result, rely on the Head pointer, which points to the first node, in order to determine the starting node.

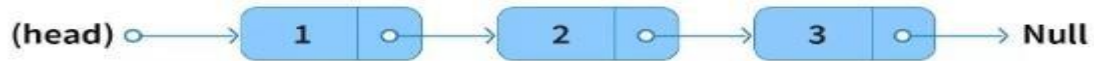
Circular linked lists save the overhead of traversing until the Node refers to null and then using a head pointer to circle back from start; however, by default, an intrinsic feature of circular linked lists provides the ability to circle the linked list.

What is a Circular-Linked-List in Data Structures?

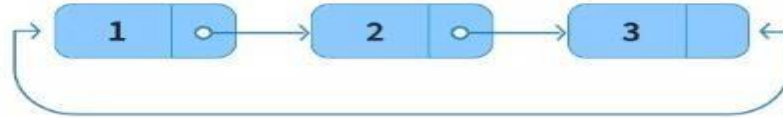
The circular linked lists are an alternative to linked lists, as we have seen. The last node in a linked-list leads to null; in circular linked-lists, link to points the last node to the fresh node. As a result, the components become connected in a circle, and the list is mention to as a circular linked-list.

A circular-linked-list can be traversed until the starting node is reached. There is no null value in the following section of any node in the circular linked list, and it has neither a beginning nor an end. It differs from the linked-list in the following ways.

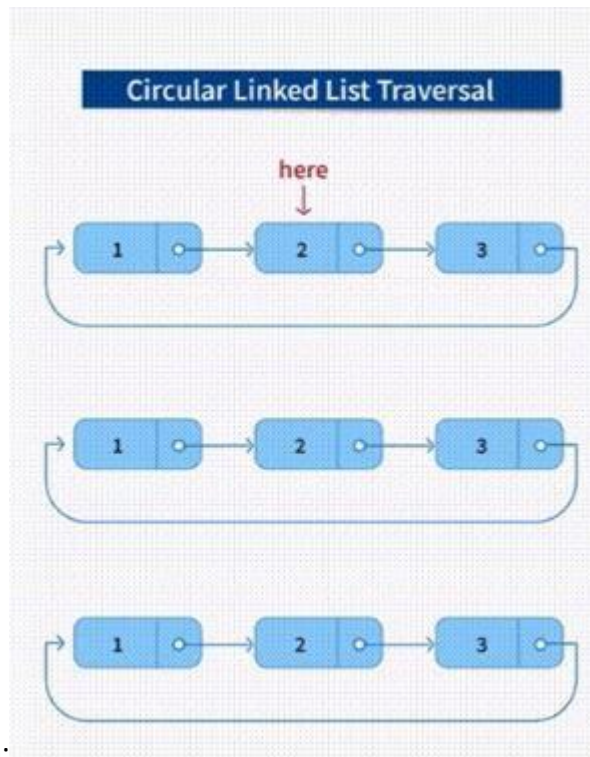
Linked List



Circular Linked List



Circular Singly Linked List



in a circular fashion:

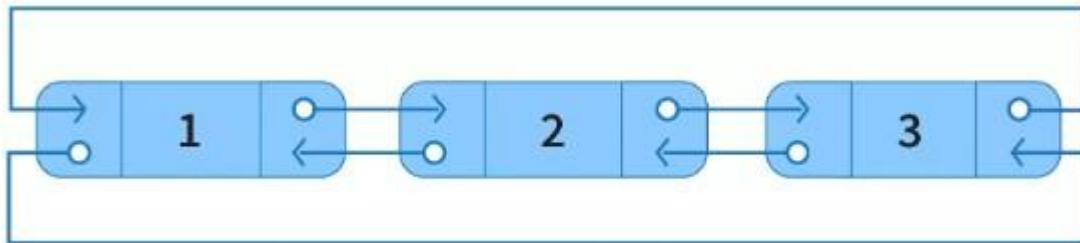
Circular Doubly-Linked-List

As shown by Linked Lists, there are numerous uses for which it is necessary to visit the list in both the ways. We own the Doubly Linked List to facilitate the extra backward transversal.

Each node in a circular singly linked-list has 2 pointers, Next and Back, that points to the node after it and the node before it, respectively, plus the next of last, which points to the node before it and vice versa, forming a circle. These lists can also make use of this property. We call

this kind of data structure a doubly-linked circular list. The appearance of the circular doubly linked list is depicted in the following image:

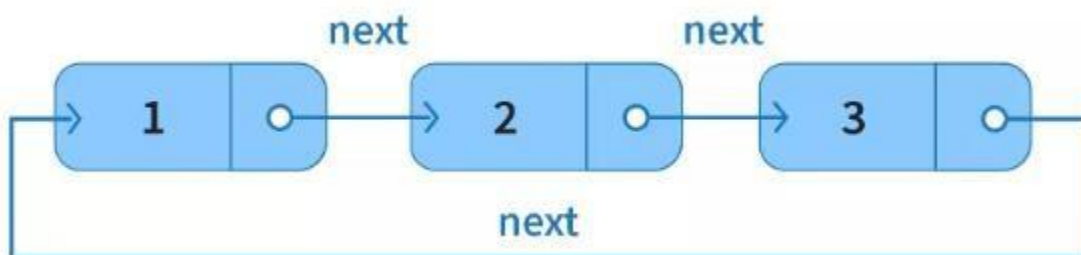
Data Structures' Representation of a Circular-Linked-List.



Node in the Circular-Linked-List can be represented as:

```
class Node{  
    int value;  
    Node next; {  
}
```

Now we will create a simple circular linked list with three Nodes to understand how this works.



This can be represented as:

// Initialize the Nodes.

```
Node one = new Node(1);  
Node two = new Node(2);  
Node three = new Node(3);
```

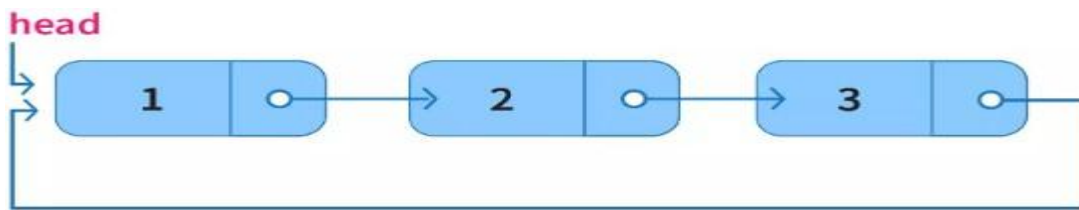
Data Structure Operations on a Circular-Linked-List

a representation of the circular-linked-list. We require a pointer to any Nodes in the Circular Linked List so that we can navigate the whole List in order to carry out additional actions on it. The head pointer, or pointer to the first node, is typically maintained in linked lists. Let's investigate whether the identical Node performs optimally in a circular-linked-list.

1. Insertion

Any new node can be added to the provided circular-linked-list.

If we maintain the Head Pointer and wish to add a new node at the start of the circular linked list, as demonstrated:



to add a new Node (let's say 4) in the start. How can the existing Nodes in the list be modified to make opportunity for this new Node?

The next of a new Node (i.e., 4) can point to Head since we maintain the head pointer (to 1), which takes care of adding 4 to the list.

But something is missing from this situation. The circular property hasn't been modified yet.

The last node, or 3, is still pointing to 1, even though 3 should now be pointing to 4 in light of the addition of 4. How is it possible to accomplish this?

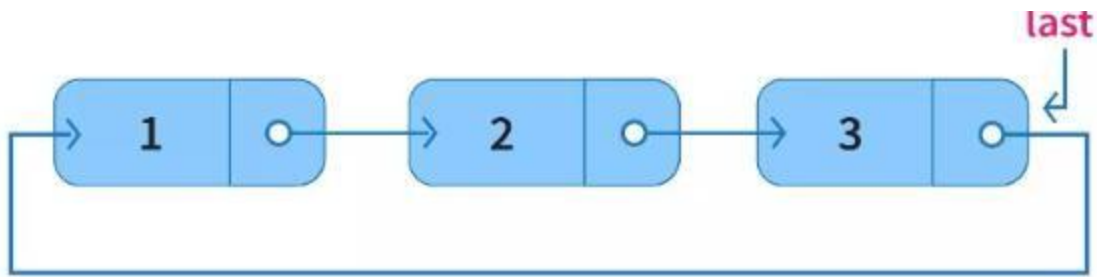
We can visit the list, get to the circular-linked-list's last node (To 3), and then move on to the next node (i.e. 4). However, the procedure is slow, as you may imagine. Imagine a situation in which we have a circular linked list with 1000 nodes, and we need to visit all 1000 nodes in order to get to the final node in order to add one.

In a similar vein, we must go through If we need to add a fresh node at the end, we should reorder the list from head to last node (after 3)

It appears that the head pointer in a data structure is not the ideal pointer for a circular linked list. What would happen, if we kept the pointer to the Last Node instead?

In both scenarios, there will be no need to go through the entire list if we use a pointer to the last node in place of a start pointer.

Using last. next, we may hop directly to head from last, thus the head pointer is not really necessary. As a result, the pointer to the final node rather than the head can be preserved. So the circular linked list in Data Structure can be visualized as:



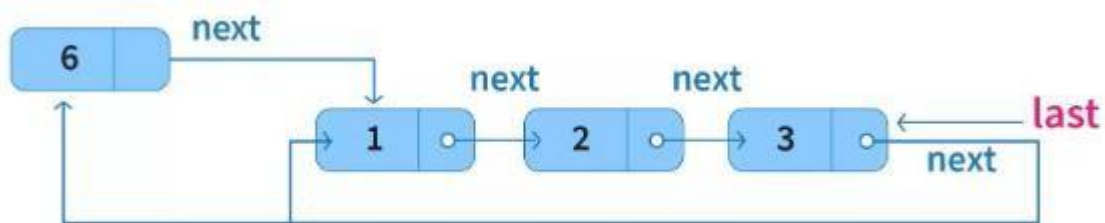
•

Insertion at the Beginning:

use Last.next to obtain the first Node directly because we have maintained the link to the Last Node.

The Node can be added at the start as:

- The New Node can point to this node as its next by obtaining the current First Node (via last.next). This fixes the New Node's pointer adjustments.
- Respect the circular characteristic by pointing the Last Node toward the New Node. This can be written programmatically as:



```

"void insertNodeAtBeginning(int value) {
Node newNode = new Node(value);
Node oldFirstNode = last.next;
newNode.next = oldFirstNode;
last.next = newNode;
}"
  
```

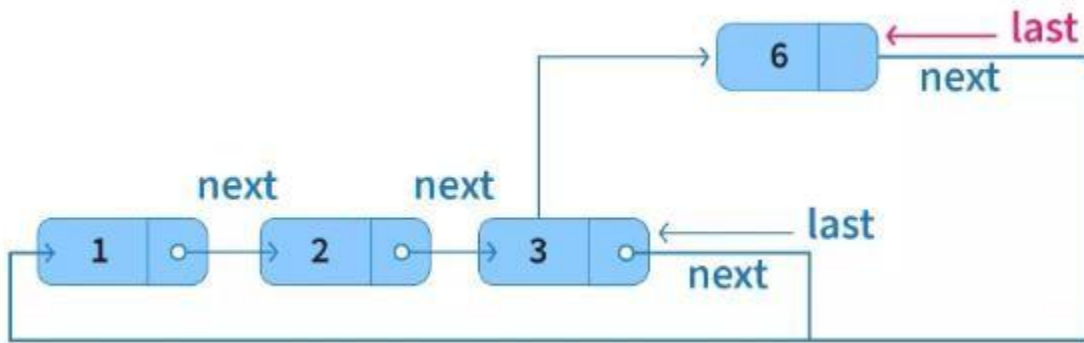
to modify the new Node's and the previous Node's references. Therefore, if we have a pointer to the last node, it requires constant time operation.

Additionally, since no additional space was utilized, the space complexness—that is, $O(1)$ space complexity—to add a new node to the start of the circular linked list stays constant.

- **Insertion at the end:**

how the New Node can be inserted in-between 2 Nodes.

1. The next of a new node will point to the last.next.
2. This new node will be pointed to by Last.next.
3. Update the last node reference to point to the new node as we now have one at the end.



This can be written as:

```

"void addAtEnd(int value) {
  Node newNode = new Node(value);

  // Adjusting the links.
  newNode.next = last.next;
  last.next = newNode;
  last = newNode;
}"
  
```

Because we just need to change the final node's pointers to add a new node to the circular linked-list, adding a fresh node at the end of the list requires constant processing time.

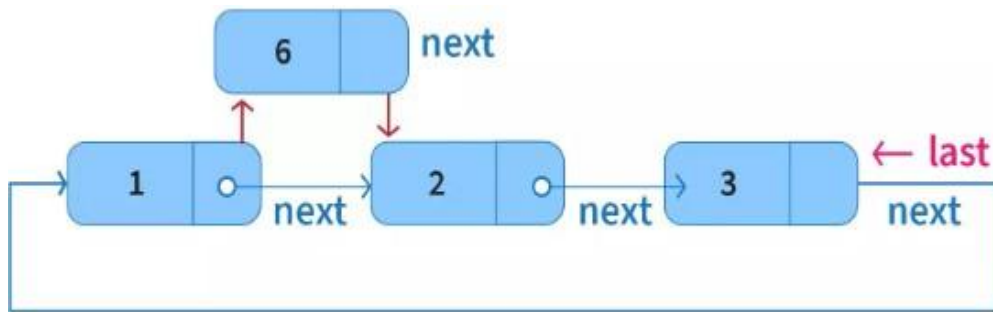
The space complexity to add a new node at the end of the circular-linked-list is $O(1)$ since no more space has been used.

- **Insertion After Another Node:**

Let's see how the New Node can be inserted in between 2 Nodes.

1. Continue navigating until we arrive at the designated node (let's say X).
2. Set the NewNode's next to the X's next.

3. Point this new Node at the X.



This can be written programmatically as:

```

void addAfter(int newValue, Node nodeBefore) {
    if (nodeBefore == null) {
        return;
    }
    Node newNode;
    Node transversalNode = last.next; // Transverse the List from last Node onwards.
    do {
        if (transversalNode.value == nodeBefore.value) { // We found the node.
            newNode = new Node(newValue);

            // Adjusting the links
            newNode.next = transversalNode.next;
            transversalNode.next = newNode;

            if (transversalNode == last) {
                // If the nodeBefore was LastNode itself, meaning we are inserting this node at the end,
                // adjust the last pointer to point to this node now.
                last = newNode;
            }
        }
        transversalNode = transversalNode.next;
        // Keep transversing until we reach the Node after which the new node has to be inserted.
    } while (transversalNode != last.next);
}
  
```

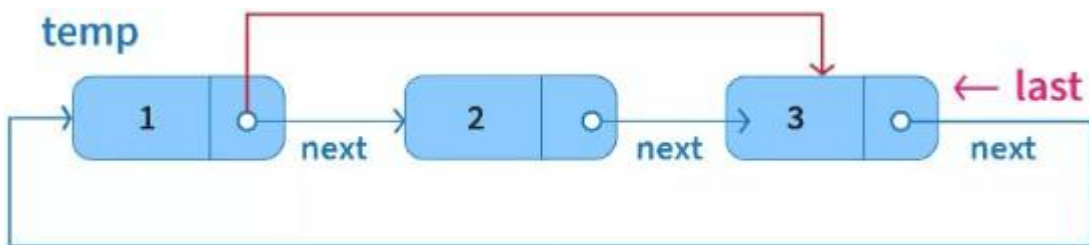
2. Deletion

- There are three possible ways to remove a specific node from a circular-linked-list:
- The CLL becomes empty if it is the only node in it. In this instance, the last node, which was pointing to the sole node, might now point to null.
 - a. Should the node that needs to be removed be the final node within the CLL

- b. To discover the node that needs to be deleted before another, iterate the circular linked list. Call it X if you will.
- c. To remove the last from the CLL, point X.next to null.
- d. Set X as the new last node by setting the last pointer to X.

1. Deleting node in the CLL:

- To discover the node to delete before another, iterate the circular linked list. Please refer to it as X.
- Give the node that has to be erased the name Y.
- X.next must be changed to point to Y.next in order to delete Y.



```

Node deleteNode(int valueOfNodeToDelete) {
    if (last == null)
        return null;
    if (last.value == valueOfNodeToDelete && last.next == last) {
        last = null;
        return last;
    }
    Node temp = last;
    if (last.value == valueOfNodeToDelete) {
        while (temp.next != last) {
            temp = temp.next;
        }
        temp.next = last.next;
        last = temp.next;
    }
    while (temp.next != last && temp.next.value != valueOfNodeToDelete) {
        temp = temp.next;
    }
    if (temp.next.value == valueOfNodeToDelete) {
        Node toDelete = temp.next;
        temp.next = toDelete.next;
    }
    return last;
}

```

3. Display

The circular linked list must be displayed by traversing from the last Pointer till we get back to it. We are able to print the Nodes' values as we traverse over them.

```
void display List() {  
    Node temp = last;  
    if (last != null) {  
        do {  
            System.out.print(temp.value + " ");  
            temp = temp.next;  
        } while (temp != last);  
    }  
}
```

Applications of Circular Linked List

When we need to preserve the circular order between the various Nodes, circular linked lists come in helpful. as circular linked lists, no Node points to null, as contrast to linked lists.

Many real-world systems use circular linked lists because they make it simple to go back to the starting node from any given one. Among these are the following:

- In games with multiple players, every player is represented by a Node in a circular-linked-list. By exploiting the circular aspect, we can quickly shuffle back to the first player from the final player, ensuring that every participant gets an opportunity to play. Operation systems apply this principle in a similar way to run many programs. These apps are all arranged in a circular linked list, so the system may quickly circle back to the beginning of the list without worrying about reaching the end of the running applications.

Summary

Two different kinds of circular linked lists exist:

1. Circular Singly Linked List: This type of list only allows one way of transverse traversal while preserving its circular nature.
2. Circular Doubly Linked Lists: these lists allow for both direction transverse traversal while preserving their circular nature.

The circular linked list's Nodes are simple to add and remove, and the process is remarkably similar to that used for linked lists. To make things easier, we use the circular-linked-lists "last" pointer rather than the linked list's "head" pointer.

1. Dynamic Data Structures

One of the primary applications of linked-lists is the creation of dynamic data structures. Unlike arrays, which have a immobile size, linked lists can grow or shrink dynamically as needed. This makes linked lists ideal for scenarios where the size of the data structure is unknown or may

change over time. For example, linked lists are commonly used to implement stacks, queues, and hash tables.

2. Memory Management

Linked lists are also used for memory management in C programs. When dynamically allocating memory using functions like malloc or calloc, linked-lists can be used to keep track of allocated memory blocks. Each node in the linked list can represent a memory block, and the pointers between nodes can be used to navigate and manage the allocated memory.

3. File Systems

Linked lists are widely used in file systems to represent directories and files. Each node in the linked list can represent a file or a directory, and the pointers between nodes can be used to establish the hierarchical structure of the file system. Linked lists provide an efficient way to traverse and manipulate the file system structure.

4. Graphs and Trees

Linked-lists are often used to utilize graphs and trees, which are essential data structures in computer science. In a graph, each node can be represented by a linked list, where each component in the linked list represents an edge connecting the node to other nodes. Similarly, in a tree, each node can be represented by a linked list, where each component in the linked list represents a child node. Linked lists provide a flexible and efficient way to represent and traverse these complex data structures.

5. Polynomial Implementation

Linked-lists are commonly used to represent polynomials in mathematics. Each node in the linked list can represent a term in the polynomial, with the coefficient and exponent stored as data in the node. The pointers between nodes can be used to establish the order of the terms in the polynomial. Linked lists provide a convenient way to perform operations on polynomials, such as addition, subtraction, and multiplication.

6. Undo/Redo Functionality

Linked-lists can be used to implement undo/redo functionality in applications. Each node in the linked list can represent a state or action, and the pointers between nodes can be used to navigate between different states. This allows users to undo or redo previous actions in an application, providing a valuable feature for user interaction.

These are just a few examples of the many applications of linked lists in the C programming language. Linked lists are versatile and can be used in various scenarios where dynamic data structures, memory management, hierarchical structures, or ordered collections are required. Understanding and mastering linked lists is essential for any programmer working with the C language.

OPERATIONS ON LINKED-LIST :

A linked-list is a data structures made up of a series of nodes, each of which has a reference (or link) to the node after it in the sequence as well as a data element. Linked lists are dynamic data structures that have the ability to alteration size as a program runs

the main operations that can be performed on a linked-list:

Insertion: Inserting a fresh node into a linked-list can be done in several ways, depending on the position where the new node needs to be inserted. The common insertion operations are:

- a) Inserting at the opening of the list: This involves creating a fresh node, setting its data, and updating the link to point to the current first node.
- b) Inserting at the end of the list: This requires traversing the list until the last node is reached, creating a new node, and updating the link of the last node to point to the new node.
- c) Inserting at a specific position: This involves traversing the list until the desired position is reached, creating a new node, and updating the links of the adjacent nodes to include the new node.

Deletion: Removes a node from a linked-list can also be done in various ways, depending on the position of the node to be deleted. The common deletion operations are:

- b) Deleting the first node: This involves updating the link of the first node to point to the second node and freeing the memory occupied by the first node.
- c) Deleting the last node: This requires traversing the linked-list further the second-to-last node is reached, updating its link to NULL, and freeing the memory inhabited by the last node.
- d) Deleting a node at a specific position: This involves traversing the list until the desired position is reached, updating the links of the adjacent nodes to bypass the node to be deleted, and freeing its memory.

Traversal: Traversing a linked-list means visiting each node in the list and performing some operation on its data. This can be done using a loop that starts from the first node and continues until the last node is reached, updating the current node reference to the next node in each iteration.

Searching: Searching for a specific value in a linked-list involves traversing the list and comparing the data of each node with the target value. If a match is found, the search operation can be terminated, and the position or existence of the value can be determined.

Updating: Updating the data of a node in a linked list can be done by traversing the list until the desired node is reached and modifying its data field.

Counting: Counting the number of nodes in a linked-list requires traversing the list and incrementing a counter variable for each node visited.

Reversing: When a linked list is reversed, the links between its nodes are altered, reversing the order such that the last node becomes the first, the second-to-last node becomes the second, and so on. one for the current node, one for the node before it, and one for the node after it.

These are the main operations that can be performed on a Each operation has its own implementation logic and complexity considerations. It is important to handle edge cases, such as empty lists or operations on the first or last node, to ensure the correct functioning of the linked list.