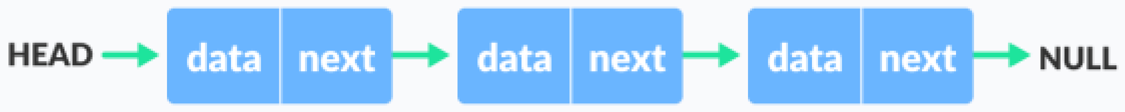
Linked list Data Structure

A linked list is a linear data structure that includes a series of connected nodes. Here, each node store the **data** and the **address** of the next node. For example,



Linked list Data Structure

You have to start somewhere, so we give the address of the first node a special name called `HEAD`. Also, the last node in the linked list can be identified because its next portion points to `NULL`.
Linked lists can be of multiple types: **singly**, **doubly**, and **circular linked list**.

Representation of Linked List

Each node consists:

- A data item
- An address of another node

Advantages of Linked List

1. The linked list is a dynamic data structure.
2. You can also decrease and increase the linked list at run-time. That is, you can allocate and deallocate memory at run-time itself.

3. In this, you can easily do insertion and deletion functions. That is, you can easily insert and delete the node
4. Memory is well utilized in the linked list. Because in it, we do not have to allocate memory in advance.
5. Its access time is very fast, and it can be accessed at a certain time without memory overhead.
6. You can easily implement linear data structures using the linked list like a stack, queue.

Disadvantages of Linked List

1. The linked list requires more memory to store the elements than an array, because each node of the linked list points a pointer, due to which it requires more memory.
2. It is very difficult to traverse the nodes in a linked list. In this, we cannot access randomly to any one node. (As we do in the array by index.) For example: – If we want to traverse a node in an n position, then we have to traverse all the nodes that come before n, which will spoil a lot of our time.
3. Reverse traversing in a linked list is very difficult, because it requires more memory for the pointer.

Application of Linked List

The linked list is a primitive data structure, which is used in various types of applications.

1. It is used to maintain directory names.
2. The linked list can perform arithmetic operations in the long integer.
3. Polynomials can be manipulated by storing constant in the node of the linked list.
4. We can also use it to next and previous images in the image viewer.
5. With the help of the linked list, we can move songs back and forth in the music player.
6. The linked list is also used for undo in word and Photoshop applications.
7. All the running applications in the computer are stored in the circular linked list, and the operating system provides them with a fixed time slot.
8. It can also be used to implement hash tables.

Difference between linked list and array

Both array and linked list are used to store the same type of linear data, but array is allocated contiguous memory location in the compile-time while the linked list is allocated memory in run-time.

Array	Linked List
It is a collection of the same type of data type.	It is a collection of similar elements that are connected to each other by the pointers.
It supports random access, which means that we can access it directly using its index, like arr[0] for 1st element, arr[7] for the 8th element, etc.	It supports sequential-access, meaning that in order to use any node in it, the user must traverse the whole list sequentially.
In the array, elements are stored in the contiguous-memory location.	The elements in the linked list can be stored anywhere in memory.
The time complexity of the array is O (1).	The time complexity of the linked list is O (n).
It is allocated the memory at compile-time.	It is allocated the memory at run-time.
Arrays take longer to perform insertion and deletion functions than linked lists.	In the linked list, both insertion and deletion operations take less time than the array.
It can be a 1-d array, 2-d array, or 3-d array.	It can be a linear linked list, doubly linked list, or circular linked list.

1. To create a new node

   ```
   Node newNode=new Node(10);
   ```

2. To add a node
3. 1Place the temp at the node after which you need to add

4. To place the temp at the last node

```
Node temp=head;
while(temp.next!=null)
    temp=temp->next;
```

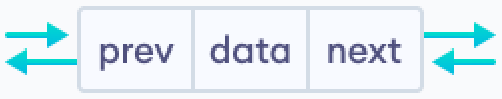
5. To traverse beyond last node

```
Node temp=head;
While(temp!=null)
    temp=temp->next
```

Doubly Linked List

A doubly linked list is a type of linked list in which each node consists of 3 components:

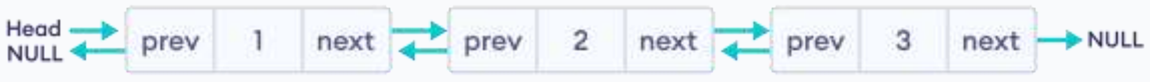
- `*prev` - address of the previous node
- `data` - data item
- `*next` - address of next node



A doubly linked list node

Representation of Doubly Linked List

Let's see how we can represent a doubly linked list on an algorithm/code. Suppose we have a doubly linked list:



Newly created doubly linked list

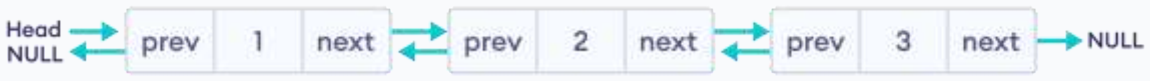
Insertion on a Doubly Linked List

Pushing a node to a doubly-linked list is similar to pushing a node to a linked list, but extra work is required to handle the pointer to the previous node.

We can insert elements at 3 different positions of a doubly-linked list:

1. Insertion at the beginning
2. Insertion in-between nodes
3. Insertion at the End

Suppose we have a double-linked list with elements **1**, **2**, and **3**.



Original doubly linked list

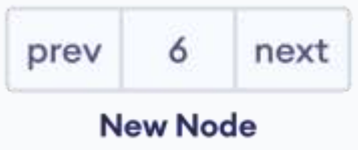
---

1. Insertion at the Beginning

Let's add a node with value **6** at the beginning of the doubly linked list we made above.

1. Create a new node

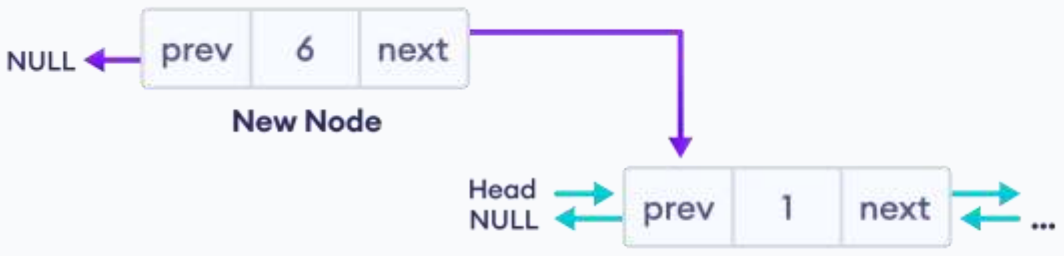
- allocate memory for `newNode`
- assign the data to `newNode`.



New node

2. Set prev and next pointers of new node

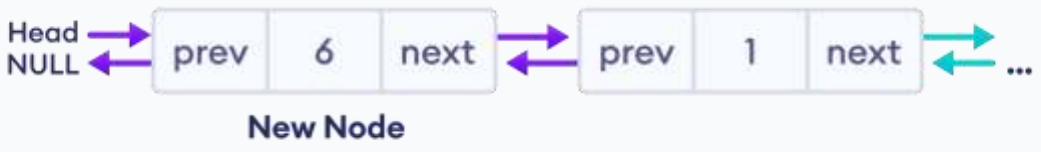
- point `next` of `newNode` to the first node of the doubly linked list
- point `prev` to `null`



Reorganize the pointers (changes are denoted by purple arrows)

3. Make new node as head node

- Point `prev` of the first node to `newNode` (now the previous `head` is the second node)
- Point `head` to `newNode`



Reorganize the pointers

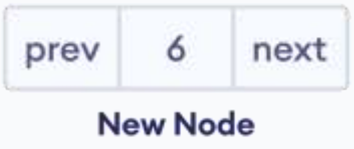
---

2. Insertion in between two nodes

Let's add a node with value 6 after node with value 1 in the doubly linked list.

1. Create a new node

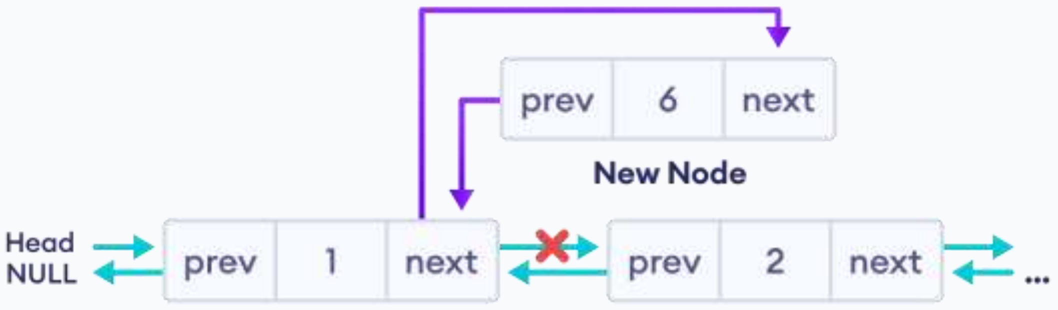
- allocate memory for `newNode`
- assign the data to `newNode`.



New node

2. Set the next pointer of new node and previous node

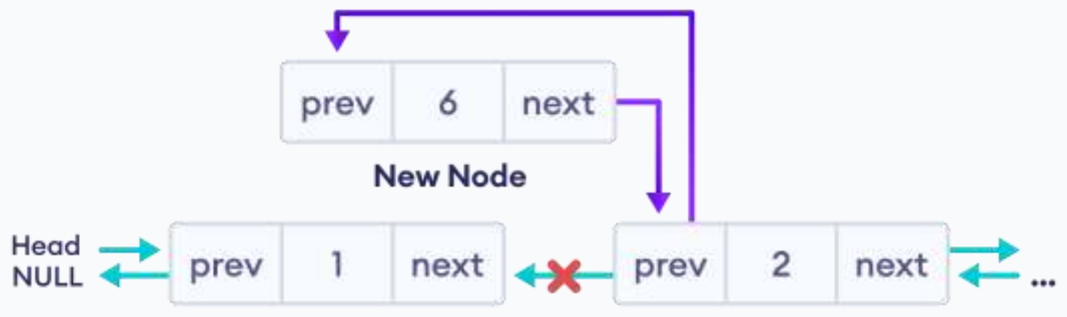
- assign the value of `next` from previous node to the `next` of `newNode`
- assign the address of `newNode` to the `next` of previous node



Reorganize the pointers

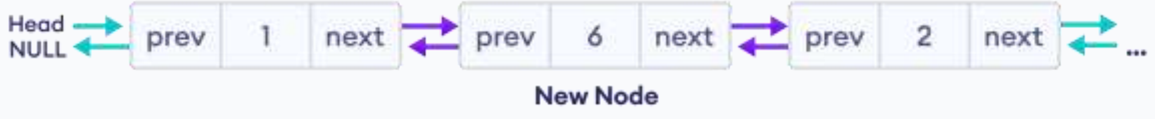
3. Set the prev pointer of new node and the next node

- assign the value of `prev` of next node to the `prev` of `newNode`
- assign the address of `newNode` to the `prev` of next node



Reorganize the pointers

The final doubly linked list is after this insertion is:

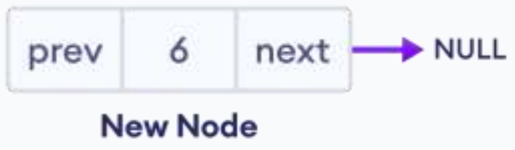


Final list

3. Insertion at the End

Let's add a node with value 6 at the end of the doubly linked list.

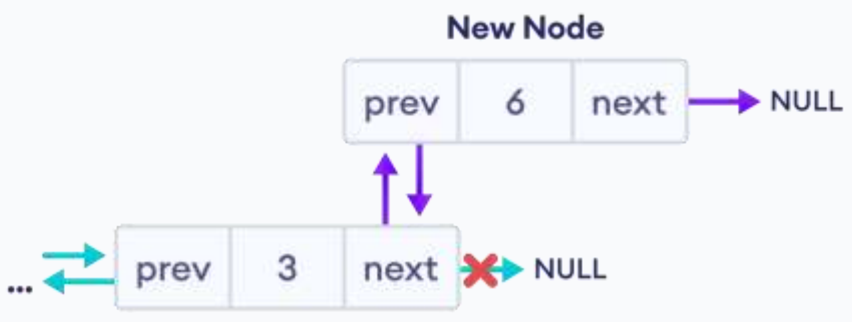
1. Create a new node



New node

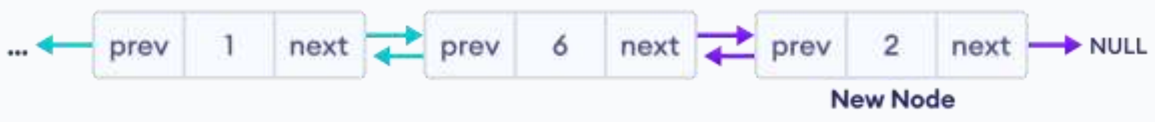
2. Set prev and next pointers of new node and the previous node

If the linked list is empty, make the `newNode` as the head node. Otherwise, traverse to the end of the doubly linked list and



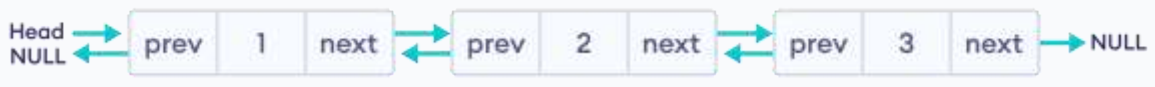
Reorganize the pointers

The final doubly linked list looks like this.



Deletion from a Doubly Linked List

Similar to insertion, we can also delete a node from **3** different positions of a doubly linked list.
Suppose we have a double-linked list with elements **1**, **2**, and **3**.



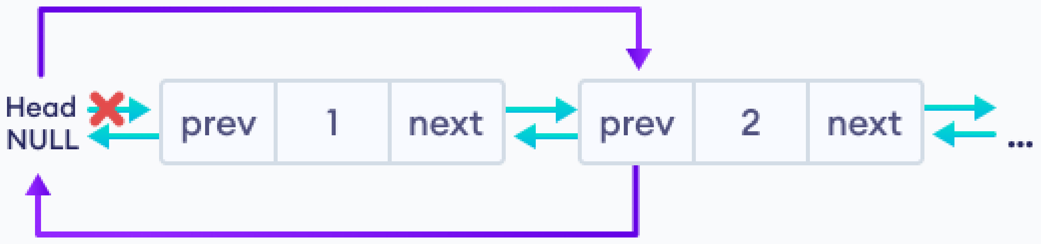
Original doubly linked list

---

1. Delete the First Node of Doubly Linked List

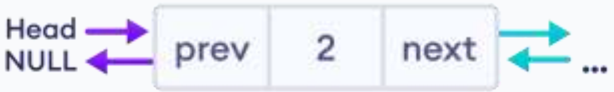
If the node to be deleted (i.e. `del_node`) is at the beginning

Reset value node after the del_node (i.e. node two)



Reorganize the pointers

Finally, free the memory of `del_node`. And, the linked will look like this



Free the space of the first node

2. Deletion of the Inner Node

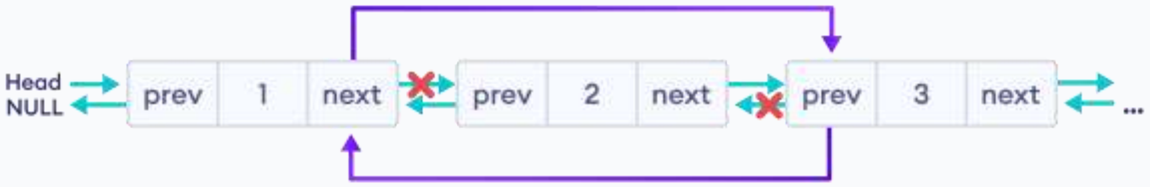
If `del_node` is an inner node (second node), we must have to reset the value of `next` and `prev` of the nodes before and after the `del_node`.

For the node before the del_node (i.e. first node)

Assign the value of `next` of `del_node` to the `next` of the `first` node.

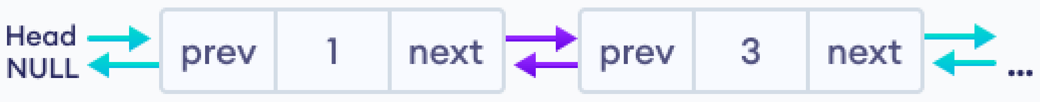
For the node after the del_node (i.e. third node)

Assign the value of `prev` of `del_node` to the `prev` of the `third` node.



Reorganize the pointers

Finally, we will free the memory of `del_node`. And, the final doubly linked list looks like this.



Final list

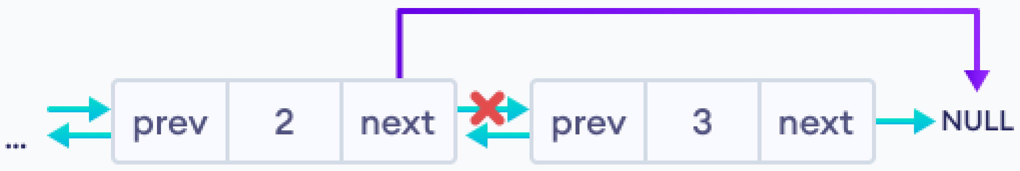
Code for Deletion of the Inner Node

```
if (del_node->next != NULL)
    del_node->next->prev = del_node->prev;

if (del_node->prev != NULL)
    del_node->prev->next = del_node->next;
```

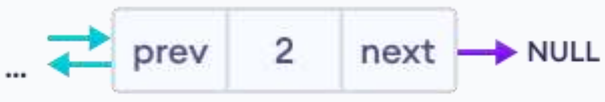
3. Delete the Last Node of Doubly Linked List

In this case, we are deleting the last node with value **3** of the doubly linked list.
Here, we can simply delete the `del_node` and make the `next` of node before `del_node` point to `NULL`.



Reorganize the pointers

The final doubly linked list looks like this.



Final list

Code for Deletion of the Last Node

```
if (del_node->prev != NULL)
```

```
    del_node->prev->next = del_node->next;
```

Here, `del_node ->next` is `NULL` so `del_node->prev->next = NULL`.

Doubly Linked List Complexity

Doubly Linked List Complexity	**Time Complexity**	**Space Complexity**
Insertion Operation	O(1) or O(n)	O(1)
Deletion Operation	O(1)	O(1)

1. Complexity of Insertion Operation

- The insertion operations that do not require traversal have the time complexity of `O(1)`.
- And, insertion that requires traversal has time complexity of `O(n)`.
- The space complexity is `O(1)`.

2. Complexity of Deletion Operation

- All deletion operations run with time complexity of `O(1)`.
- And, the space complexity is `O(1)`.

Doubly Linked List Applications

1. Redo and undo functionality in software.
2. Forward and backward navigation in browsers.

3. For navigation systems where forward and backward navigation is required.

Singly Linked List Vs Doubly Linked List

Singly Linked List	Doubly Linked List
Each node consists of a data value and a pointer to the next node.	Each node consists of a data value, a pointer to the next node and a pointer to the previous node.
Traversal can occur in one way only (forward direction).	Traversal can occur in both ways.
It requires less space.	It requires more space because of an extra pointer.
It can be implemented on the stack.	It has multiple usages. It can be implemented on the stack, heap, and binary tree.

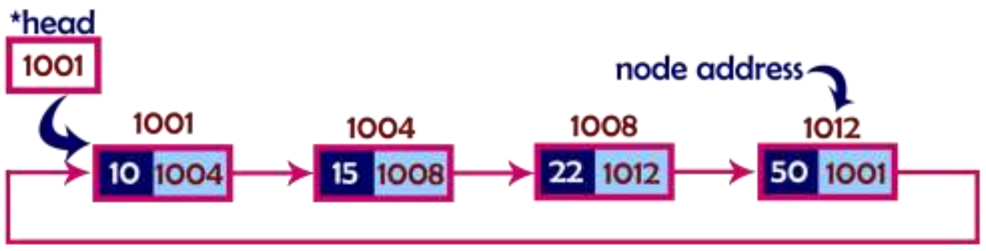
What is Circular Linked List?

In single linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list.

A circular linked list is a sequence of elements in which every element has a link to its next element in the sequence and the last element has a link to the first element.

That means circular linked list is similar to the single linked list except that the last node points to the first node in the list

Example



Operations

In a circular linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program.
- **Step 2 -** Declare all the **user defined** functions.
- **Step 3 -** Define a **Node** structure with two members **data** and **next**
- **Step 4 -** Define a Node pointer '**head**' and set it to **NULL**.
- **Step 5 -** Implement the **main** method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

In a circular linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the circular linked list...

- **Step 1 -** Create a **newNode** with given value.

- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode→next** = **head** .
- **Step 4 -** If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.
- **Step 5 -** Keep moving the '**temp**' to its next node until it reaches to the last node (until '**temp** → **next** == **head**').
- **Step 6 -** Set '**newNode** → **next** =**head**', '**head** = **newNode**' and '**temp** → **next** = **head**'.

Inserting At End of the list

We can use the following steps to insert a new node at end of the circular linked list...

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

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the circular linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode** → **next** = **head**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 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).
- **Step 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.
- **Step 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).
- **Step 8 -** If **temp** is last node then set **temp** → **next** = **newNode** and **newNode** → **next** = **head**.
- **Step 8 -** If **temp** is not last node then set **newNode** → **next** = **temp** → **next** and **temp** → **next** = **newNode**.

Deletion

In a circular linked list, the deletion operation can be performed in three ways those are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list

3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the circular linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.
- **Step 4 -** Check whether list is having only one node (**temp1** → **next** == **head**)
- **Step 5 -** If it is **TRUE** then set **head** = **NULL** and delete **temp1** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1** → **next** == **head**)
- **Step 7 -** Then set **head** = **temp2** → **next**, **temp1** → **next** = **head** and delete **temp2**.

Deleting from End of the list

We can use the following steps to delete a node from end of the circular linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.
- **Step 4 -** Check whether list has only one Node (**temp1** → **next** == **head**)
- **Step 5 -** If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)
- **Step 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**)
- **Step 7 -** Set **temp2** → **next** = **head** and delete **temp1**.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the circular linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 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.

- **Step 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.
- **Step 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**)
- **Step 7 -** If list has only one node and that is the node to be deleted then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
- **Step 8 -** If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).
- **Step 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**.
- **Step 10 -** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).
- **Step 1 1-** If **temp1** is last node then set **temp2** → **next** = **head** and delete **temp1** (**free(temp1)**).
- **Step 12 -** If **temp1** is not first node and not last node then set **temp2** → **next** = **temp1** → **next** and delete **temp1** (**free(temp1)**).

Displaying a circular Linked List

We can use the following steps to display the elements of a circular linked list...

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

Advantage of circular linked list

- Entire list can be traversed from any node of the list.
- It saves time when we have to go to the first node from the last node.
- Its is used for the implementation of queue.
- Reference to previous node can easily be found.
- When we want a list to be accessed in a circle or loop then circular linked list are used.

Disadvantage of circular linked list

- Circular list are complex as compared to singly linked lists.
- Reversing of circular list is a complex as compared to singly or doubly lists.
- If not traversed carefully, then we could end up in an infinite loop.

- Like singly and doubly lists circular linked lists also doesn’t support direct accessing of elements.

Applications of Circular Linked List

A Circular Linked List can be used for the following –

- Circular lists are used in applications where the entire list is accessed one-by-one in a loop.
- It is also used by the Operating system to share time for different users, generally uses a Round-Robin time-sharing mechanism.
- Multiplayer games use a circular list to swap between players in a loop.
- Implementation of Advanced data structures like Fibonacci Heap
- The browser cache which allows you to hit the BACK button
- Undo functionality in Photoshop or Word
- Circular linked lists are used in Round Robin Scheduling
- Circular linked list used Most recent list (MRU LIST)