UNIT-III

LINKED LIST

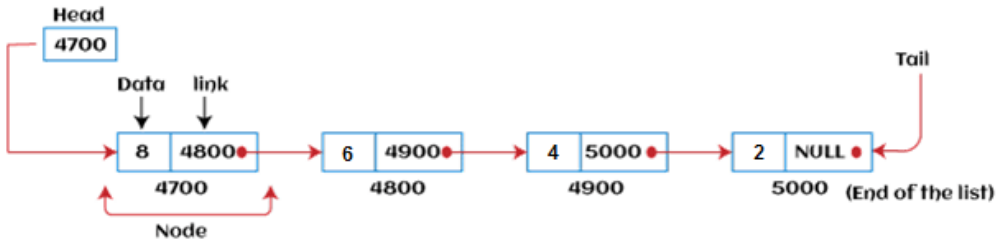
Linked list is a linear data structure that includes a series of connected nodes. Linked list can be defined as the nodes that are randomly stored in the memory.

A node in the linked list contains two parts, i.e., first is the data part and second is the address part. The last node of the list contains a pointer to the null.

After array, linked list is the second most used data structure. In a linked list, every link contains a connection to another link.

Representation of a Linked list

Linked list can be represented as the connection of nodes in which each node points to the next node of the list. The representation of the linked list is shown below -



Why use linked list over array?

Linked list is a data structure that overcomes the limitations of arrays. Let's first see some of the limitations of arrays -

- The size of the array must be known in advance before using it in the program.
- Increasing the size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
- All the elements in the array need to be contiguously stored in the memory. Inserting an element in the array needs shifting of all its predecessors.

Linked list is useful because -

- It allocates the memory dynamically. All the nodes of the linked list are non-contiguously stored in the memory and linked together with the help of pointers.

- In linked list, size is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

Types of Linked list

Linked list is classified into the following types -

- **Singly-linked list -** Singly linked list can be defined as the collection of an ordered set of elements. A node in the singly linked list consists of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node, while the link part of the node stores the address of its immediate successor.
- **Doubly linked list -** Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly-linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer), and pointer to the previous node (previous pointer).
- **Circular singly linked list -** In a circular singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.
- **Circular doubly linked list -** Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the nodes. The last node of the list contains the address of the first node of the list. The first node of the list also contains the address of the last node in its previous pointer.

Now, let's see the benefits and limitations of using the Linked list.

Advantages of Linked list

The advantages of using the Linked list are given as follows -

- **Dynamic data structure -** The size of the linked list may vary according to the requirements. Linked list does not have a fixed size.
- **Insertion and deletion -** Unlike arrays, insertion, and deletion in linked list is easier. Array elements are stored in the consecutive location, whereas the elements in the linked list are stored at a random location. To insert or delete

an element in an array, we have to shift the elements for creating the space. Whereas, in linked list, instead of shifting, we just have to update the address of the pointer of the node.

- **Memory efficient -** The size of a linked list can grow or shrink according to the requirements, so memory consumption in linked list is efficient.
- **Implementation -** We can implement both stacks and queues using linked list.

Disadvantages of Linked list

The limitations of using the Linked list are given as follows -

- **Memory usage -** In linked list, node occupies more memory than array. Each node of the linked list occupies two types of variables, i.e., one is a simple variable, and another one is the pointer variable.
- **Traversal -** Traversal is not easy in the linked list. If we have to access an element in the linked list, we cannot access it randomly, while in case of array we can randomly access it by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.
- **Reverse traversing -** Backtracking or reverse traversing is difficult in a linked list. In a doubly-linked list, it is easier but requires more memory to store the back pointer.

Applications of Linked list

The applications of the Linked list are given as follows -

- With the help of a linked list, the polynomials can be represented as well as we can perform the operations on the polynomial.
- A linked list can be used to represent the sparse matrix.
- The various operations like student's details, employee's details, or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.
- Using linked list, we can implement stack, queue, tree, and other various data structures.
- The graph is a collection of edges and vertices, and the graph can be represented as an adjacency matrix and adjacency list. If we want to represent the graph as an adjacency matrix, then it can be implemented as

an array. If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.

- A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.

Operations performed on Linked list

The basic operations that are supported by a list are mentioned as follows -

- **Insertion -** This operation is performed to add an element into the list.
- **Deletion -** It is performed to delete an operation from the list.
- **Display -** It is performed to display the elements of the list.
- **Search -** It is performed to search an element from the list using the given key.

Representation:

A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head points to NULL.

Each node in a list consists of at least two parts:

1) data (we can store integer, strings or any type of data).

2) Pointer (Or Reference) to the next node (connects one node to another)

In python, LinkedList can be represented as a class and a Node as a separate class. The LinkedList class contains a reference of Node class type.

```
# Node class

class Node:

    # Function to initialize the node object
    def __init__(self, data):
        self.data = data  # Assign data
        self.next = None  # Initialize
                          # next as null

# Linked List class
class LinkedList:

    # Function to initialize the Linked
    # List object
    def __init__(self):
        self.head = None
```

SINGLY LINKED LIST OR ONE WAY CHAIN:

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program.

A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.

In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject.

The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

SN	Operation	Description

1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

Deletion and Traversing

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

SN	Operation	Description
1	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
2	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
3	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.
4	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.

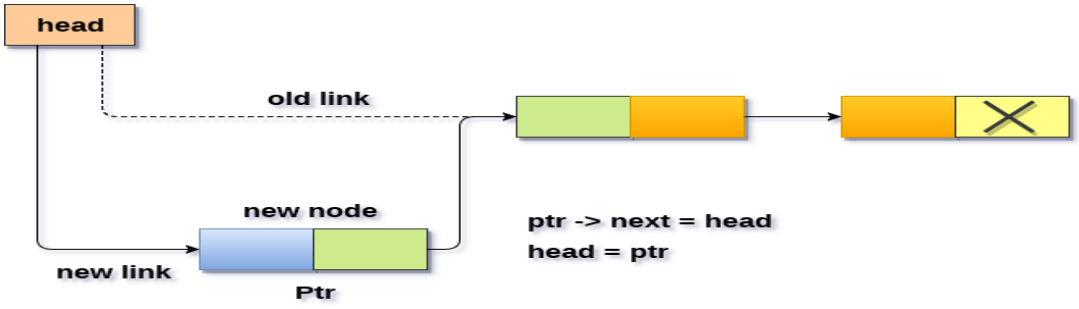
5	Searching	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. .

Insertion in singly linked list at beginning:

Inserting a new element into a singly linked list at beginning is quite simple. We just need to make a few adjustments in the node links.

- Allocate the space for the new node and store data into the data part of the node.
- Make the link part of the new node pointing to the existing first node of the list.
- At the last, we need to make the new node as the first node of the list this will be done by using the following statement.

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



Algorithm for insertion at end of the list:

In order to insert a node at the last, there are two following scenarios which need to be mentioned.

1. The node is being added to an empty list
2. The node is being added to the end of the linked list

in the first case,

The condition (head == NULL) gets satisfied. Hence, we just need to allocate the space for the node.

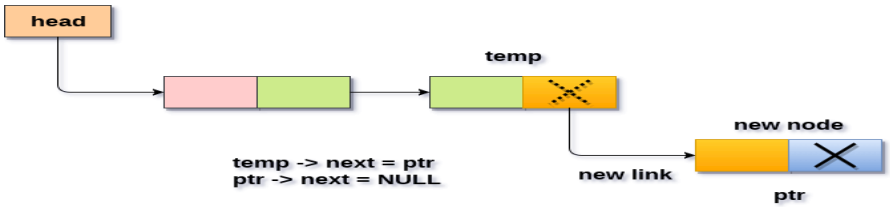
Since, **ptr** is the only node that will be inserted in the list hence, we need to make this node pointed by the head pointer of the list.

In the second case,

- The condition **Head = NULL** would fail, since Head is not null. Now, we need to declare a temporary pointer temp in order to traverse through the list. **temp** is made to point the first node of the list.

- Then, traverse through the entire linked list.
- At the end of the loop, the temp will be pointing to the last node of the list. Now, allocate the space for the new node, and assign the item to its data part.
- Since, the new node is going to be the last node of the list hence, the next part of this node needs to be pointing to the **null**.
- We need to make the next part of the temp node (which is currently the last node of the list) point to the new node (ptr) .

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



Inserting node at the last into a non-empty list

Algorithm for insertion after specified node:

In order to insert an element after the specified number of nodes into the linked list, we need to skip the desired number of elements in the list to move the pointer at the position after which the node will be inserted.

Allocate the space for the new node and add the item to the data part of it.

Now, we just need to make a few more link adjustments and our node at will be inserted at the specified position. Since, at the end of the loop, the loop pointer temp would be pointing to the node after which the new node will be inserted.

Therefore, the next part of the new node ptr must contain the address of the next part of the temp (since, ptr will be in between temp and the next of the temp).

Now, we just need to make the next part of the temp, point to the new node ptr. This will insert the new node ptr, at the specified position.

STEP 1: IF PTR = NULL
WRITE OVERFLOW
GOTO STEP 12
END OF IF
STEP 2: SET NEW_NODE = PTR
STEP 3: NEW_NODE → DATA = VAL
STEP 4: SET TEMP = HEAD
STEP 5: SET I = 0
STEP 6: REPEAT STEP 5 AND 6 UNTIL I<loc< li="">
STEP 7: TEMP = TEMP → NEXT
STEP 8: IF TEMP = NULL
WRITE "DESIRED NODE NOT PRESENT"
GOTO STEP 12

END OF IF

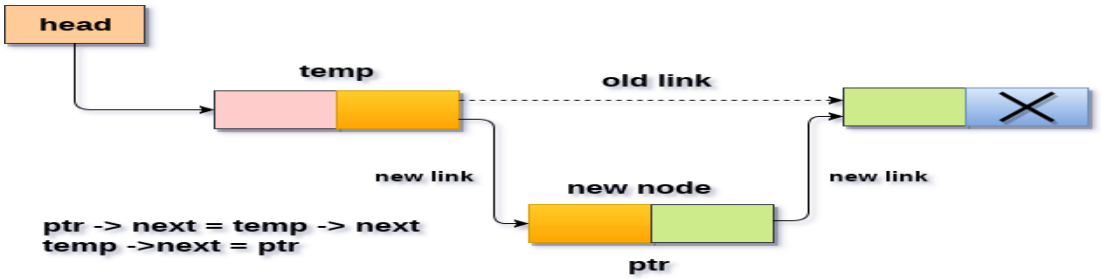
END OF LOOP

STEP 9: PTR → NEXT = TEMP → NEXT

STEP 10: TEMP → NEXT = PTR

STEP 11: SET PTR = NEW_NODE

STEP 12: EXIT




```
# A complete working Python program to demonstrate all
# insertion methods of linked list

# Node class
class Node:

    # Function to initialise the node object
    def __init__(self, data):
        self.data = data  # Assign data
        self.next = None  # Initialize next as null


# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None


    # Function to insert a new node at the beginning
    def push(self, new_data):

        # 1 & 2: Allocate the Node &
        #        Put in the data
        new_node = Node(new_data)

        # 3. Make next of new Node as head
        new_node.next = self.head
```

```
    # 4. Move the head to point to new Node
    self.head = new_node


# This function is in LinkedList class. Inserts a
# new node after the given prev_node. This method is
# defined inside LinkedList class shown above */
def insertAfter(self, prev_node, new_data):

    # 1. check if the given prev_node exists
    if prev_node is None:
        print("The given previous node must inLinkedList.")
        return

    #  2. create new node &
    #      Put in the data
    new_node = Node(new_data)

    # 4. Make next of new Node as next of prev_node
    new_node.next = prev_node.next

    # 5. make next of prev_node as new_node
    prev_node.next = new_node


# This function is defined in Linked List class
# Appends a new node at the end.  This method is
# defined inside LinkedList class shown above */
def append(self, new_data):

    # 1. Create a new node
    # 2. Put in the data
    # 3. Set next as None
    new_node = Node(new_data)

    # 4. If the Linked List is empty, then make the
    #    new node as head
    if self.head is None:
        self.head = new_node
        return

    # 5. Else traverse till the last node
    last = self.head
    while (last.next):
        last = last.next
```

```
        # 6. Change the next of last node
        last.next =  new_node


    # Utility function to print the linked list
    def printList(self):
        temp = self.head
        while (temp):
            print(temp.data,end=" ")
            temp = temp.next



# Code execution starts here
if __name__=='__main__':

    # Start with the empty list
    llist = LinkedList()

    # Insert 6.  So linked list becomes 6->None
    llist.append(6)

    # Insert 7 at the beginning. So linked list becomes 7->6->None
    llist.push(7);

    # Insert 1 at the beginning. So linked list becomes 1->7->6->None
    llist.push(1);

    # Insert 4 at the end. So linked list becomes 1->7->6->4->None
    llist.append(4)

    # Insert 8, after 7. So linked list becomes 1 -> 7-> 8-> 6-> 4-> None
    llist.insertAfter(llist.head.next, 8)

    print('Created linked list is: ')
    llist.printList()
```

Output:

```
Created Linked list is:  1  7  8  6  4
```

<u>Deletion in singly linked list at beginning:</u>

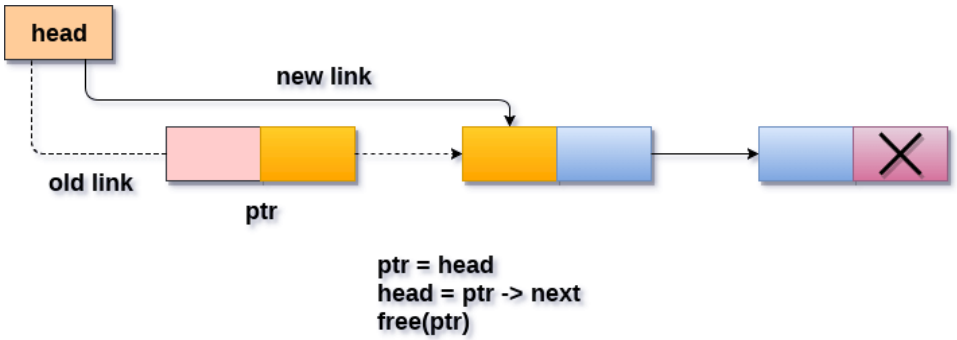
Deleting a node from the beginning of the list is the simplest operation of all. It just need a few adjustments in the node pointers. Since the first node of the list is

to be deleted, therefore, we just need to make the head, point to the next of the head.

Now, free the pointer ptr which was pointing to the head node of the list.

Algorithm

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



Deleting a node from the beginning

Deletion in singly linked list at the end:

There are two scenarios in which, a node is deleted from the end of the linked list.

1. There is only one node in the list and that needs to be deleted.
2. There are more than one node in the list and the last node of the list will be deleted.

In the first scenario,

The condition head → next = NULL will survive and therefore, the only node head of the list will be assigned to null.

In the second scenario,

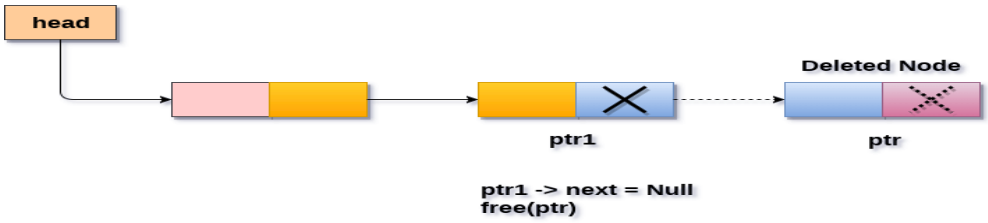
The condition head → next = NULL would fail and therefore, we have to traverse the node in order to reach the last node of the list.

For this purpose, just declare a temporary pointer temp and assign it to head of the list. We also need to keep track of the second last node of the list. For this purpose, two pointers ptr and ptr1 will be used where ptr will point to the last node and ptr1 will point to the second last node of the list.

Now, we just need to make the pointer ptr1 point to the NULL and the last node of the list that is pointed by ptr will become free.

Algorithm

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



Deleting a node from the last

Deletion in singly linked list after the specified node:

In order to delete the node, which is present after the specified node, we need to skip the desired number of nodes to reach the node after which the node will be deleted. We need to keep track of the two nodes.

The one which is to be deleted the other one if the node which is present before that node. For this purpose, two pointers are used: ptr and ptr1.

Now, our task is almost done, we just need to make a few pointer adjustments. Make the next of ptr1 (points to the specified node) point to the next of ptr (the node which is to be deleted).

Algorithm

STEP 1: IF HEAD = NULL

WRITE UNDERFLOW

GOTO STEP 10

END OF IF

STEP 2: SET TEMP = HEAD

STEP 3: SET I = 0

STEP 4: REPEAT STEP 5 TO 8 UNTIL I<loc< li="">

STEP 5: TEMP1 = TEMP

STEP 6: TEMP = TEMP → NEXT

STEP 7: IF TEMP = NULL

WRITE "DESIRED NODE NOT PRESENT"

GOTO STEP 12

END OF IF

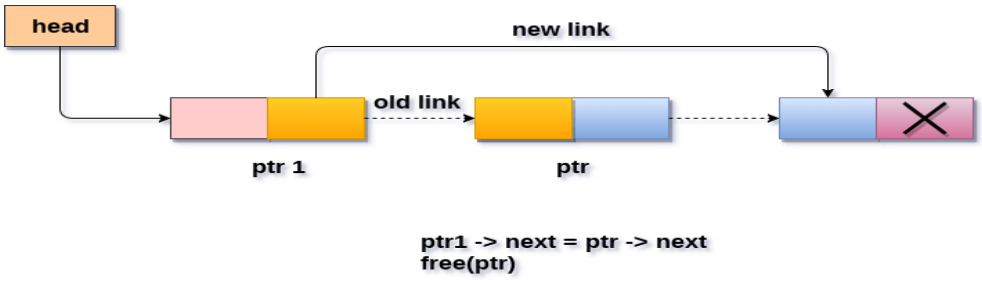
STEP 8: I = I+1

END OF LOOP

STEP 9: TEMP1 → NEXT = TEMP → NEXT

STEP 10: FREE TEMP

STEP 11: EXIT



Deletion a node from specified position

Traversing in singly linked list:

Traversing is the most common operation that is performed in almost every scenario of singly linked list. Traversing means visiting each node of the list once in order to perform some operation on that. This will be done by using the following statements.

Algorithm

STEP 1: SET PTR = HEAD
STEP 2: IF PTR = NULL
WRITE "EMPTY LIST"
GOTO STEP 7
END OF IF
STEP 4: REPEAT STEP 5 AND 6 UNTIL PTR != NULL
STEP 5: PRINT PTR→ DATA
STEP 6: PTR = PTR → NEXT
[END OF LOOP]
STEP 7: EXIT

Searching in singly linked list:

Searching is performed in order to find the location of a particular element in the list. Searching any element in the list needs traversing through the list and make the comparison of every element of the list with the specified element.

If the element is matched with any of the list element then the location of the element is returned from the function.

Algorithm

Step 1: SET PTR = HEAD
Step 2: Set I = 0
STEP 3: IF PTR = NULL
WRITE "EMPTY LIST"
GOTO STEP 8
END OF IF
STEP 4: REPEAT STEP 5 TO 7 UNTIL PTR != NULL
STEP 5: if ptr → data = item

write i+1

End of IF

STEP 6: I = I + 1

STEP 7: PTR = PTR → NEXT

[END OF LOOP]

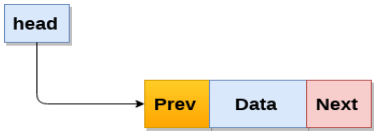
STEP 8: EXIT

Doubly linked list

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence.

Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer).

A sample node in a doubly linked list is shown in the figure.



Node

A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.

Doubly Linked List

The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

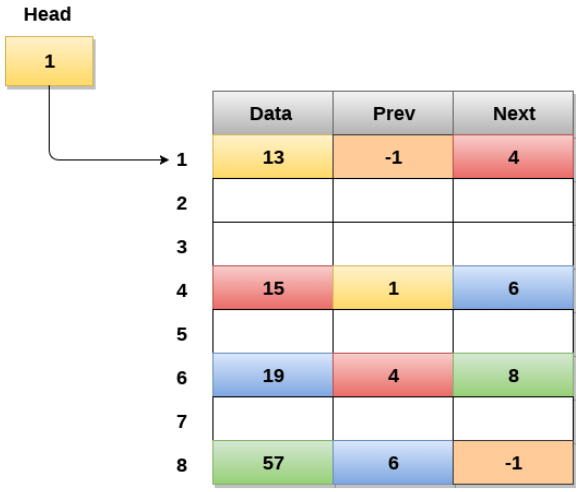
In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes.

However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Memory Representation of a doubly linked list:

Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion.

However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).



Memory Representation of a Doubly linked list

Operations on doubly linked list:

Node representation of a doubly linked list

```
# Node of a doubly linked list
class Node:
   def __init__(self, next=None, prev=None, data=None):
      self.next = next # reference to next node in DLL
      self.prev = prev # reference to previous node in DLL
      self.data = data
```

All the remaining operations regarding doubly linked list are described in the following table.

SN	Operation	Description

1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion after specified node	Adding the node into the linked list after the specified node.
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Deletion of the node having given data	Removing the node which is present just after the node containing the given data.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

Insertion in doubly linked list at beginning:

As in doubly linked list, each node of the list contain double pointers therefore we have to maintain more number of pointers in doubly linked list as compare to singly linked list.

There are two scenarios of inserting any element into doubly linked list. Either the list is empty or it contains at least one element. Perform the following steps to insert a node in doubly linked list at beginning.

- Allocate the space for the new node in the memory.
- Check whether the list is empty or not. The list is empty if the condition head == NULL holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.

- In the second scenario, the condition **head == NULL** become false and the node will be inserted in beginning. The next pointer of the node will point to the existing head pointer of the node. The prev pointer of the existing head will point to the new node being inserted.

Since, the node being inserted is the first node of the list and therefore it must contain NULL in its prev pointer. Hence assign null to its previous part and make the head point to this node.

Algorithm :

Step 1: IF ptr = NULL
Write OVERFLOW
Go to Step 8
[END OF IF]

Step 2: SET NEW_NODE = ptr

Step 3: SET NEW_NODE -> DATA = VAL

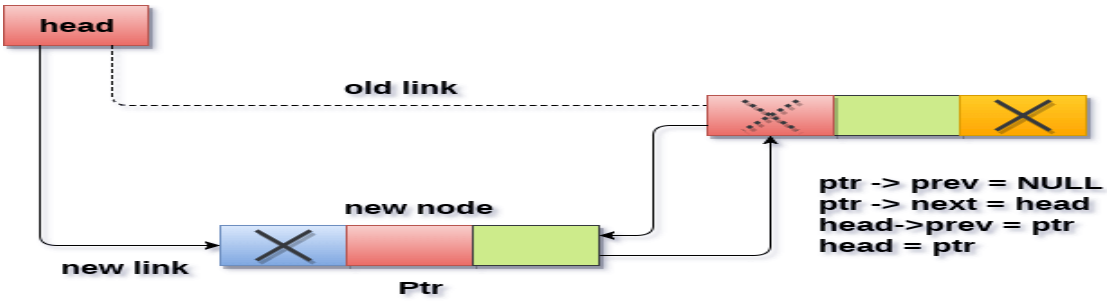
Step 4: SET NEW_NODE -> PREV = NULL

Step 5: SET NEW_NODE -> NEXT = START

Step 6: SET head -> PREV = NEW_NODE

Step 7: SET head = NEW_NODE

Step 8: EXIT



Insertion into doubly linked list at beginning

Insertion in doubly linked list at the end:

In order to insert a node in doubly linked list at the end, we must make sure whether the list is empty or it contains any element. Use the following steps in order to insert the node in doubly linked list at the end.

- Allocate the memory for the new node. Make the pointer **ptr** point to the new node being inserted.

- Check whether the list is empty or not. The list is empty if the condition **head == NULL** holds. In that case, the node will be inserted as the only node of the

list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.

- In the second scenario, the condition head == NULL become false. The new node will be inserted as the last node of the list. For this purpose, we have to traverse the whole list in order to reach the last node of the list. Initialize the pointer **temp** to head and traverse the list by using this pointer.

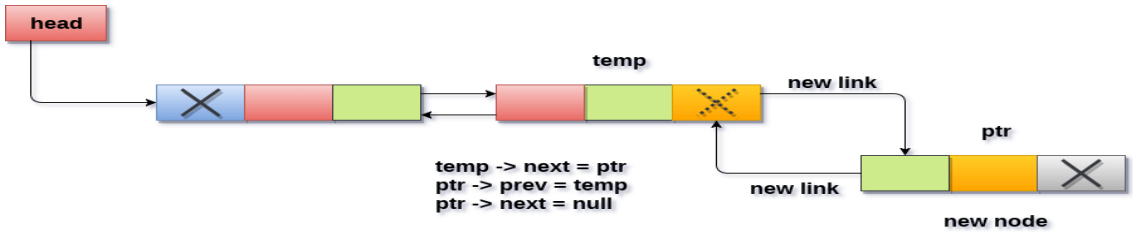
The pointer temp point to the last node at the end of this while loop. Now, we just need to make a few pointer adjustments to insert the new node ptr to the list. First, make the next pointer of temp point to the new node being inserted i.e. ptr.

make the previous pointer of the node ptr point to the existing last node of the list i.e. temp.

make the next pointer of the node ptr point to the null as it will be the new last node of the list.

Algorithm:

Step 1: IF PTR = NULL
Write OVERFLOW
Go to Step 10
[END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET NEW_NODE -> DATA = VAL
Step 4: SET NEW_NODE -> NEXT = NULL
Step 5: SET TEMP = HEAD
Step 6: Repeat Step 7 while TEMP -> NEXT != NULL
Step 7: SET TEMP = TEMP -> NEXT
[END OF LOOP]
Step 8: SET TEMP -> NEXT = NEW_NODE
Step 9: SET NEW_NODE -> PREV = TEMP
Step 10: EXIT



Insertion into doubly linked list at the end

Insertion in doubly linked list after Specified node:

In order to insert a node after the specified node in the list, we need to skip the required number of nodes in order to reach the mentioned node and then make the pointer adjustments as required.

Use the following steps for this purpose.

- Allocate the memory for the new node.
- Traverse the list by using the pointer **temp** to skip the required number of nodes in order to reach the specified node.
- The temp would point to the specified node at the end of the **for** loop. The new node needs to be inserted after this node therefore we need to make a few pointer adjustments here. Make the next pointer of **ptr** point to the next node of temp.

make the **next** pointer of temp point to the new node ptr.

Algorithm:

Step 1: IF PTR = NULL
Write OVERFLOW
Go to Step 14
[END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET NEW_NODE -> DATA = VAL
Step 4: SET TEMP = head
Step 5: SET I = 0
Step 6: REPEAT 7 to 9 until I
Step 7: SET TEMP = TEMP -> NEXT

STEP 8: IF TEMP = NULL

STEP 9: WRITE "LESS THAN DESIRED NO. OF ELEMENTS"

GOTO STEP 14

[END OF IF]

[END OF LOOP]

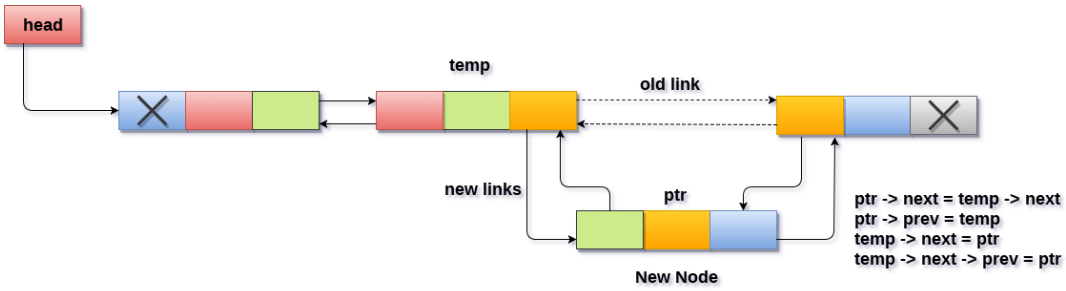
Step 10: SET NEW_NODE -> NEXT = TEMP -> NEXT

Step 11: SET NEW_NODE -> PREV = TEMP

Step 12 : SET TEMP -> NEXT = NEW_NODE

Step 13: SET TEMP -> NEXT -> PREV = NEW_NODE

Step 14: EXIT



Insertion into doubly linked list after specified node

Deletion at beginning:

Deletion in doubly linked list at the beginning is the simplest operation. We just need to copy the head pointer to pointer ptr and shift the head pointer to its next.

Now make the prev of this new head node point to NULL. This will be done by using the following statements.

Now free the pointer ptr by using the **free** function.

Algorithm

STEP 1: IF HEAD = NULL

WRITE UNDERFLOW

GOTO STEP 6

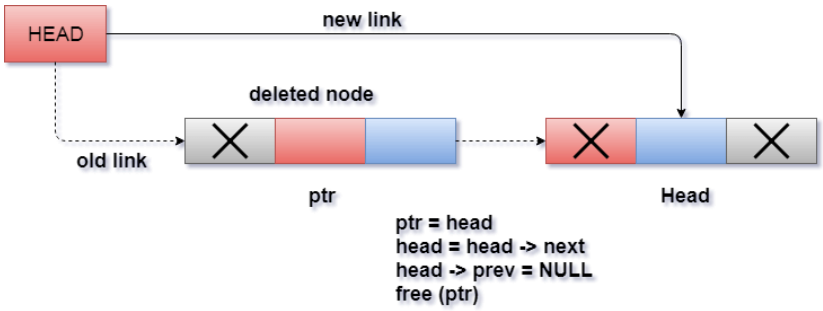
STEP 2: SET PTR = HEAD

STEP 3: SET HEAD = PTR→ NEXT

STEP 4: SET HEAD → PREV = NULL

STEP 5: FREE PTR

STEP 6: EXIT



Deletion in doubly linked list from beginning

Deletion in doubly linked list at the end:

Deletion of the last node in a doubly linked list needs traversing the list in order to reach the last node of the list and then make pointer adjustments at that position.

In order to delete the last node of the list, we need to follow the following steps.

- If the list is already empty then the condition head == NULL will become true and therefore the operation can not be carried on.
- If there is only one node in the list then the condition head → next == NULL become true. In this case, we just need to assign the head of the list to NULL and free head in order to completely delete the list.
- Otherwise, just traverse the list to reach the last node of the list. This will be done by using the following statements.

- The ptr would point to the last node of the ist at the end of the for loop. Just make the next pointer of the previous node of **ptr** to **NULL**.

free the pointer as this the node which is to be deleted.

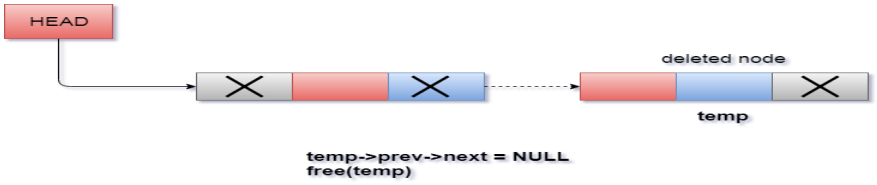
Algorithm

Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 7
[END OF IF]
Step 2: SET TEMP = HEAD
Step 3: REPEAT STEP 4 WHILE TEMP->NEXT != NULL
Step 4: SET TEMP = TEMP->NEXT
[END OF LOOP]

Step 5: SET TEMP ->PREV-> NEXT = NULL

Step 6: FREE TEMP

Step 7: EXIT



Deletion in doubly linked list at the end

Deletion in doubly linked list after the specified node:

In order to delete the node after the specified data, we need to perform the following steps.

- Copy the head pointer into a temporary pointer temp.
- Traverse the list until we find the desired data value.
- Check if this is the last node of the list. If it is so then we can't perform deletion.
- Check if the node which is to be deleted, is the last node of the list, if it so then we have to make the next pointer of this node point to null so that it can be the new last node of the list.
- Otherwise, make the pointer ptr point to the node which is to be deleted. Make the next of temp point to the next of ptr. Make the previous of next node of ptr point to temp. free the ptr.

Algorithm:

Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 9
[END OF IF]

Step 2: SET TEMP = HEAD

Step 3: Repeat Step 4 while TEMP -> DATA != ITEM

Step 4: SET TEMP = TEMP -> NEXT
[END OF LOOP]

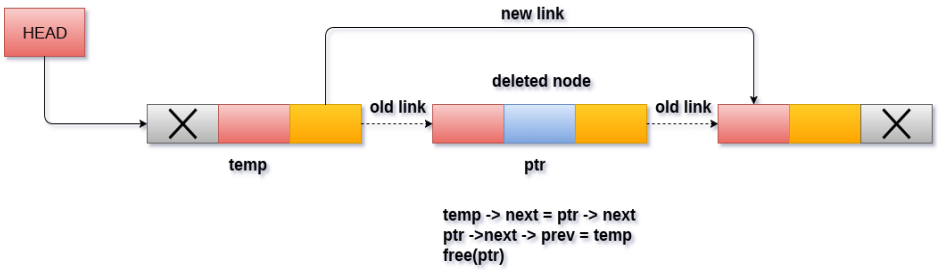
Step 5: SET PTR = TEMP -> NEXT

Step 6: SET TEMP -> NEXT = PTR -> NEXT

Step 7: SET PTR -> NEXT -> PREV = TEMP

Step 8: FREE PTR

Step 9: EXIT



Deletion of a specified node in doubly linked list

Searching for a specific node in Doubly Linked List:

We just need traverse the list in order to search for a specific element in the list. Perform following operations in order to search a specific operation.

- Copy head pointer into a temporary pointer variable ptr.
- declare a local variable I and assign it to 0.
- Traverse the list until the pointer ptr becomes null. Keep shifting pointer to its next and increasing i by +1.
- Compare each element of the list with the item which is to be searched.
- If the item matched with any node value then the location of that value I will be returned from the function else NULL is returned.

Algorithm:

Step 1: IF HEAD == NULL
WRITE "UNDERFLOW"
GOTO STEP 8
[END OF IF]

Step 2: Set PTR = HEAD

Step 3: Set i = 0

Step 4: Repeat step 5 to 7 while PTR != NULL

Step 5: IF PTR → data = item
return i
[END OF IF]

Step 6: i = i + 1

Step 7: PTR = PTR → next

Step 8: Exit

Traversing in doubly linked list:

Traversing is the most common operation in case of each data structure. For this purpose, copy the head pointer in any of the temporary pointer ptr.

Then, traverse through the list by using while loop. Keep shifting value of pointer variable **ptr** until we find the last node. The last node contains **null** in its next part.

Although, traversing means visiting each node of the list once to perform some specific operation. Here, we are printing the data associated with each node of the list.

Algorithm:

Step 1: IF HEAD == NULL
WRITE "UNDERFLOW"
GOTO STEP 6
[END OF IF]

Step 2: Set PTR = HEAD

Step 3: Repeat step 4 and 5 while PTR != NULL

Step 4: Write PTR → data

Step 5: PTR = PTR → next

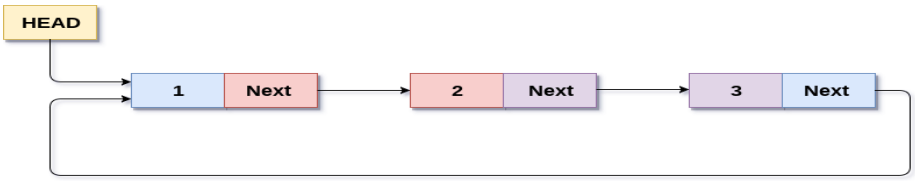
Step 6: Exit

Circular Singly Linked List:

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



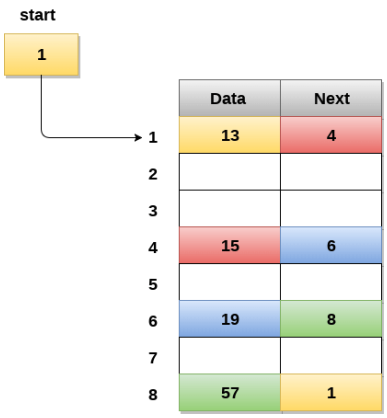
Circular Singly Linked List

Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

Memory Representation of circular linked list:

In the following image, memory representation of a circular linked list containing marks of a student in 4 subjects.

However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.



Memory Representation of a circular linked list

Operations on Circular Singly linked list:

Insertion

SN	Operation	Description
1	Insertion at beginning	Adding a node into circular singly linked list at the beginning.
2	Insertion at the end	Adding a node into circular singly linked list at the end.

Deletion & Traversing

SN	Operation	Description
1	Deletion at beginning	Removing the node from circular singly linked list at the beginning.
2	Deletion at the end	Removing the node from circular singly linked list at the end.
3	Searching	Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.
4	Traversing	Visiting each element of the list at least once in order to perform some specific operation.

Insertion into circular singly linked list at beginning

There are two scenarios in which a node can be inserted in circular singly linked list at beginning. Either the node will be inserted in an empty list or the node is to be inserted in an already filled list.

In the first scenario, the condition **head == NULL** will be true. Since, the list in which, we are inserting the node is a circular singly linked list, therefore the only node of the list (which is just inserted into the list) will point to itself only. We also need to make the head pointer point to this node.

In the second scenario, the condition **head == NULL** will become false which means that the list contains at least one node. In this case, we need to traverse the list in order to reach the last node of the list.

At the end of the loop, the pointer temp would point to the last node of the list. Since, in a circular singly linked list, the last node of the list contains a pointer to the first node of the list.

Therefore, we need to make the next pointer of the last node point to the head node of the list and the new node which is being inserted into the list will be

the new head node of the list therefore the next pointer of temp will point to the new node ptr.

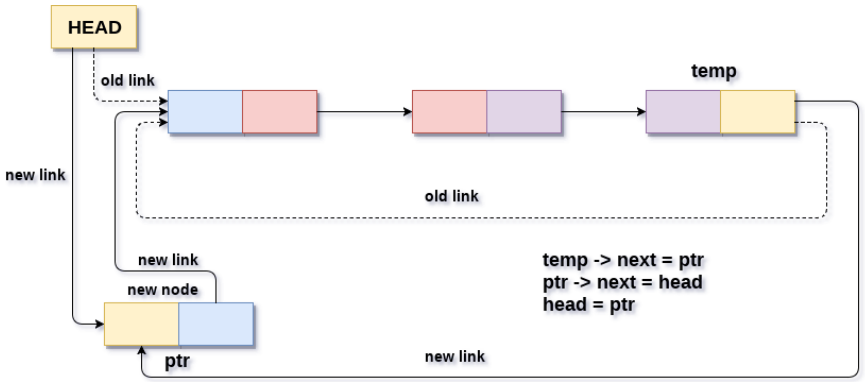
The next pointer of temp will point to the existing head node of the list.

Now, make the new node ptr, the new head node of the circular singly linked list.

in this way, the node ptr has been inserted into the circular singly linked list at beginning.

Algorithm

Step 1: IF PTR = NULL
Write OVERFLOW
Go to Step 10
[END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET NEW_NODE -> DATA = VAL
Step 4: SET TEMP = HEAD
Step 5: Repeat Step 8 while TEMP -> NEXT != HEAD
Step 6: SET TEMP = TEMP -> NEXT
[END OF LOOP]
Step 7: SET NEW_NODE -> NEXT = HEAD
Step 8: SET TEMP → NEXT = NEW_NODE
Step 9: SET HEAD = NEW_NODE
Step 10: EXIT



Insertion into circular singly linked list at beginning

Insertion into circular singly linked list at the end

There are two scenarios in which a node can be inserted in circular singly linked list at beginning. Either the node will be inserted in an empty list or the node is to be inserted in an already filled list.

In the first scenario, the condition **head == NULL** will be true. Since, the list in which, we are inserting the node is a circular singly linked list, therefore the only node of the list (which is just inserted into the list) will point to itself only.

We also need to make the head pointer point to this node. In the second scenario, the condition **head == NULL** will become false which means that the list contains at least one node.

In this case, we need to traverse the list in order to reach the last node of the list. This will be done by using the following statement.

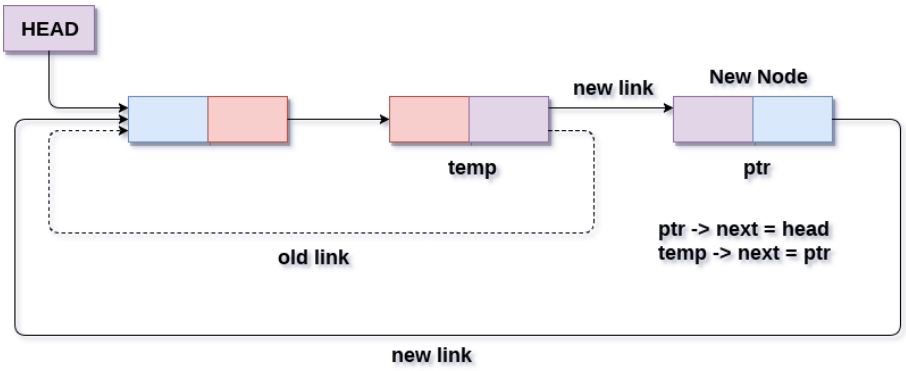
At the end of the loop, the pointer temp would point to the last node of the list. Since, the new node which is being inserted into the list will be the new last node of the list. Therefore the existing last node i.e. **temp** must point to the new node **ptr**.

The new last node of the list i.e. ptr will point to the head node of the list.

In this way, a new node will be inserted in a circular singly linked list at the beginning.

Algorithm

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



Insertion into circular singly linked list at end

Deletion in circular singly linked list at beginning:

In order to delete a node in circular singly linked list, we need to make a few pointer adjustments.

There are three scenarios of deleting a node from circular singly linked list at beginning.

Scenario 1: (The list is Empty)

If the list is empty then the condition **head == NULL** will become true, in this case, we just need to print **underflow** on the screen and make exit.

Scenario 2: (The list contains single node)

If the list contains single node then, the condition **head → next == head** will become true. In this case, we need to delete the entire list and make the head pointer free.

Scenario 3: (The list contains more than one node)

If the list contains more than one node then, in that case, we need to traverse the list by using the pointer **ptr** to reach the last node of the list.

At the end of the loop, the pointer ptr point to the last node of the list. Since, the last node of the list points to the head node of the list. Therefore this will be changed as now, the last node of the list will point to the next of the head node.

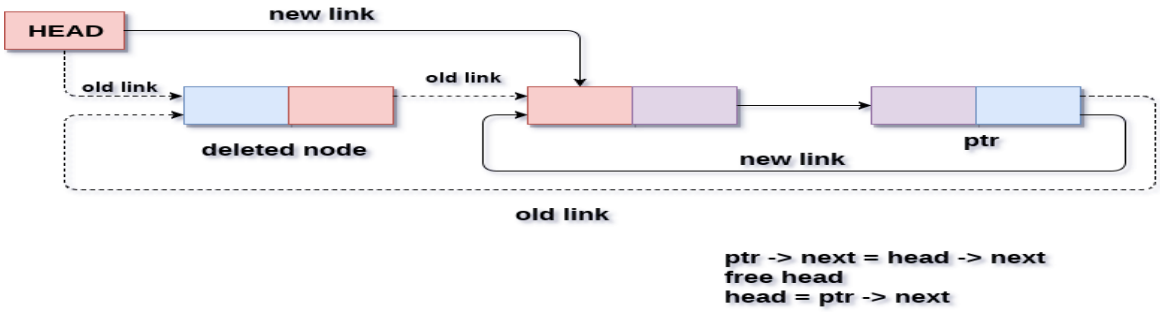
Now, free the head pointer by using the free() method in C language.

Make the node pointed by the next of the last node, the new head of the list.

In this way, the node will be deleted from the circular singly linked list from the beginning.

Algorithm

Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 8
[END OF IF]
Step 2: SET PTR = HEAD
Step 3: Repeat Step 4 while PTR → NEXT != HEAD
Step 4: SET PTR = PTR → next
[END OF LOOP]
Step 5: SET PTR → NEXT = HEAD → NEXT
Step 6: FREE HEAD
Step 7: SET HEAD = PTR → NEXT
Step 8: EXIT



Deletion in circular singly linked list at beginning

Deletion in Circular singly linked list at the end

There are three scenarios of deleting a node in circular singly linked list at the end.

Scenario 1 (the list is empty)

If the list is empty then the condition **head == NULL** will become true, in this case, we just need to print **underflow** on the screen and make exit.

Scenario 2(the list contains single element)

If the list contains single node then, the condition **head → next == head** will become true. In this case, we need to delete the entire list and make the head pointer free.
if(head->next == head)

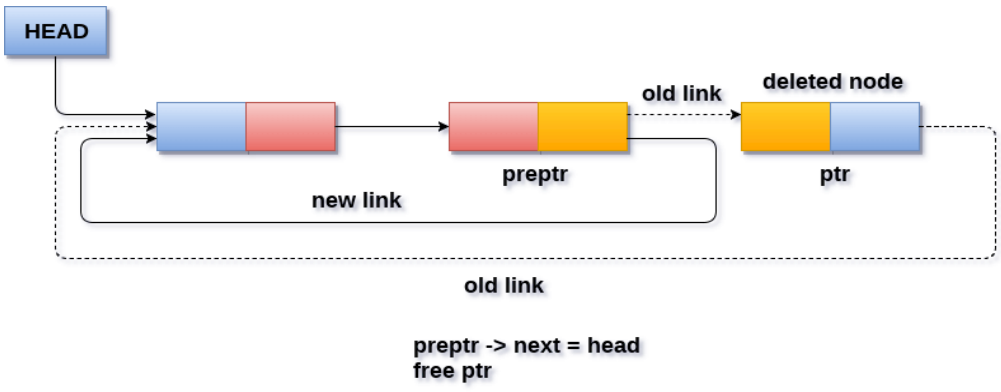
Scenario 3(the list contains more than one element)

If the list contains more than one element, then in order to delete the last element, we need to reach the last node. We also need to keep track of the second last node of the list. For this purpose, the two pointers ptr and preptr are defined.

Now, we need to make just one more pointer adjustment. We need to make the next pointer of preptr point to the next of ptr (i.e. head) and then make pointer ptr free.

Algorithm

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 != HEAD
Step 4: SET PREPTR = PTR
Step 5: SET PTR = PTR -> NEXT
[END OF LOOP]
Step 6: SET PREPTR -> NEXT = HEAD
Step 7: FREE PTR
Step 8: EXIT



Deletion in circular singly linked list at end

Searching in circular singly linked list:

Searching in circular singly linked list needs traversing across the list. The item which is to be searched in the list is matched with each node data of the list once and if the match found then the location of that item is returned otherwise -1 is returned.

Algorithm

Step 1: SET PTR = HEAD

Step 2: Set I = 0

STEP 3: IF PTR = NULL

WRITE "EMPTY LIST"

GOTO STEP 9

END OF IF

STEP 4: IF HEAD → DATA = ITEM

WRITE i+1 RETURN [END OF IF]

STEP 5: REPEAT STEP 5 TO 7 UNTIL PTR->next != head

STEP 6: if ptr → data = item

write i+1

RETURN

End of IF

STEP 7: I = I + 1

STEP 8: PTR = PTR → NEXT

[END OF LOOP]

STEP 9: EXIT

Traversing in Circular Singly linked list:

Traversing in circular singly linked list can be done through a loop. Initialize the temporary pointer variable **temp** to head pointer and run the while loop until the next pointer of temp becomes **head**. The algorithm and the c function implementing the algorithm is described as follows.

Algorithm

STEP 1: SET PTR = HEAD

STEP 2: IF PTR = NULL

WRITE "EMPTY LIST"

GOTO STEP 8

END OF IF

STEP 4: REPEAT STEP 5 AND 6 UNTIL PTR → NEXT != HEAD

STEP 5: PRINT PTR → DATA

STEP 6: PTR = PTR → NEXT

[END OF LOOP]

STEP 7: PRINT PTR→ DATA

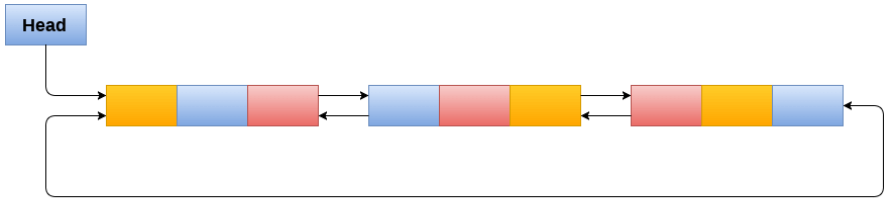
STEP 8: EXIT

CIRCULAR DOUBLY LINKED LIST:

Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node.

Circular doubly linked list doesn't contain NULL in any of the node. The last node of the list contains the address of the first node of the list. The first node of the list also contains address of the last node in its previous pointer.

A circular doubly linked list is shown in the following figure.



Circular Doubly Linked List

Due to the fact that a circular doubly linked list contains three parts in its structure therefore, it demands more space per node and more expensive basic operations. However, a circular doubly linked list provides easy manipulation of the pointers and the searching becomes twice as efficient.

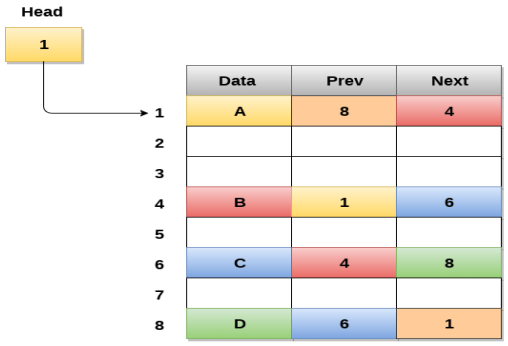
Memory Management of Circular Doubly linked list:

The following figure shows the way in which the memory is allocated for a circular doubly linked list. The variable head contains the address of the first element of the list i.e. 1 hence the starting node of the list contains data A is stored at address 1.

Since, each node of the list is supposed to have three parts therefore; the starting node of the list contains address of the last node i.e. 8 and the next node i.e. 4.

The last node of the list that is stored at address 8 and containing data as 6, contains address of the first node of the list as shown in the image.

In circular doubly linked list, the last node is identified by the address of the first node which is stored in the next part of the last node therefore the node which contains the address of the first node, is actually the last node of the list.



Memory Representation of a Circular Doubly linked list

Operations on circular doubly linked list:

There are various operations which can be performed on circular doubly linked list. The node structure of a circular doubly linked list is similar to doubly linked list. However, the operations on circular doubly linked list are described in the following table.

SN	Operation	Description
1	Insertion at beginning	Adding a node in circular doubly linked list at the beginning.
2	Insertion at end	Adding a node in circular doubly linked list at the end.
3	Deletion at beginning	Removing a node in circular doubly linked list from beginning.
4	Deletion at end	Removing a node in circular doubly linked list at the end.

Traversing and searching in circular doubly linked list is similar to that in the circular singly linked list.

Insertion in circular doubly linked list at beginning

There are two scenario of inserting a node in circular doubly linked list at beginning. Either the list is empty or it contains more than one element in the list.

Allocate the memory space for the new node **ptr** by using the following statement.

In the first case, the condition **head == NULL** becomes true therefore; the node will be added as the first node in the list. The next and the previous pointer of this newly added node will point to itself only.

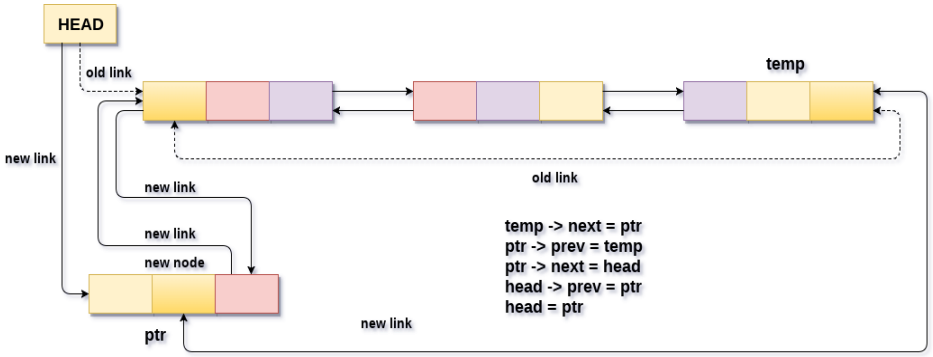
In the second scenario, the condition **head == NULL** becomes false. In this case, we need to make a few pointer adjustments at the end of the list. For this purpose, we need to reach the last node of the list through traversing the list.

At the end of loop, the pointer temp would point to the last node of the list. Since the node which is to be inserted will be the first node of the list therefore, temp must contain the address of the new node ptr into its next part.

In this way, the new node is inserted into the list at the beginning.

Algorithm:

Step 1: IF PTR = NULL
Write OVERFLOW
Go to Step 12
[END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET NEW_NODE -> DATA = VAL
Step 4: SET TEMP = HEAD
Step 5: Repeat Step 6 while TEMP -> NEXT != HEAD
Step 6: SET TEMP = TEMP -> NEXT
[END OF LOOP]
Step 7: SET TEMP -> NEXT = NEW_NODE
Step 8: SET NEW_NODE -> PREV = TEMP
Step 9 : SET NEW_NODE -> NEXT = HEAD
Step 10: SET HEAD -> PREV = NEW_NODE
Step 11: SET HEAD = NEW_NODE
Step 12: EXIT



Insertion into circular doubly linked list at beginning

Insertion in circular doubly linked list at end:

There are two scenario of inserting a node in circular doubly linked list at the end. Either the list is empty or it contains more than one element in the list.

Allocate the memory space for the new node **ptr**

In the first case, the condition **head == NULL** becomes true therefore, the node will be added as the first node in the list. The next and the previous pointer of this newly added node will point to itself only.

In the second scenario, the condition **head == NULL** become false, therefore node will be added as the last node in the list.

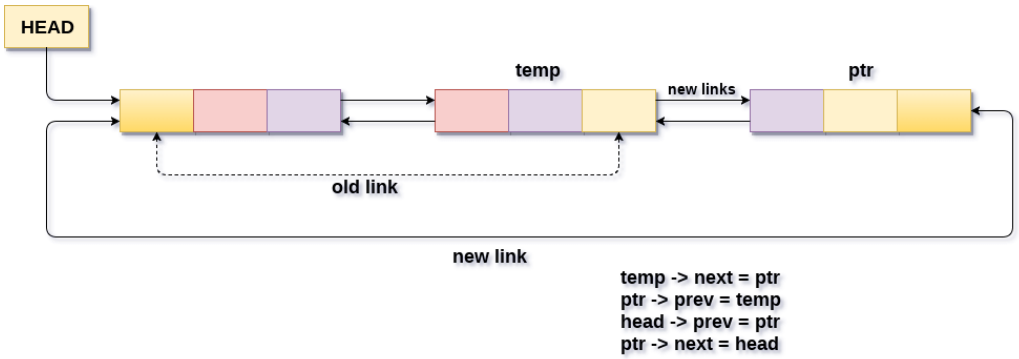
For this purpose, we need to make a few pointer adjustments in the list at the end. Since, the new node will contain the address of the first node of the list therefore we need to make the next pointer of the last node point to the head node of the list.

Similarly, the previous pointer of the head node will also point to the new last node of the list.

Now, we also need to make the next pointer of the existing last node of the list (temp) point to the new last node of the list, similarly, the new last node will also contain the previous pointer to the temp.

Algorithm

Step 1: IF PTR = NULL
Write OVERFLOW
Go to Step 11
[END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET NEW_NODE -> DATA = VAL
Step 4: SET NEW_NODE -> NEXT = HEAD
Step 5: SET TEMP = HEAD
Step 6: Repeat Step 7 while TEMP -> NEXT != HEAD
Step 7: SET TEMP = TEMP -> NEXT
[END OF LOOP]
Step 8: SET TEMP -> NEXT = NEW_NODE
Step 9: SET NEW_NODE -> PREV = TEMP
Step 10: SET HEAD -> PREV = NEW_NODE
Step 11: EXIT



Insertion into circular doubly linked list at end

Deletion in Circular doubly linked list at beginning:

There can be two scenario of deleting the first node in a circular doubly linked list.

The node which is to be deleted can be the only node present in the linked list. In this case, the condition head → next == head will become true, therefore the list needs to be completely deleted.

It can be simply done by assigning head pointer of the list to null and free the head pointer.

In the second scenario, the list contains more than one element in the list, therefore the condition **head → next == head** will become false. Now, reach the last node of the list and make a few pointer adjustments there. Run a while loop for this purpose

Now, temp will point to the last node of the list. The first node of the list i.e. pointed by head pointer, will need to be deleted.

Therefore the last node must contain the address of the node that is pointed by the next pointer of the existing head node.

The new head node i.e. next of existing head node must also point to the last node of the list through its previous pointer.

Now, free the head pointer and the make its next pointer, the new head node of the list.

Algorithm

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 9

[END OF IF]

Step 2: SET TEMP = HEAD

Step 3: Repeat Step 4 while TEMP -> NEXT != HEAD

Step 4: SET TEMP = TEMP -> NEXT

[END OF LOOP]

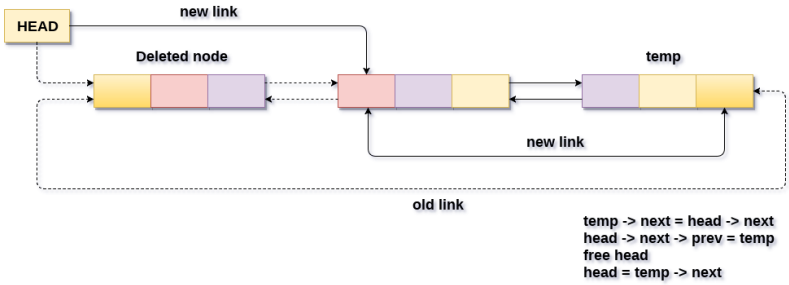
Step 5: SET TEMP -> NEXT = HEAD -> NEXT

Step 6: SET HEAD -> NEXT -> PREV = TEMP

Step 7: FREE HEAD

Step 8: SET HEAD = TEMP -> NEXT

Step 9: Exit



Deletion in circular doubly linked list at beginning

Deletion in circular doubly linked list at end:

There can be two scenario of deleting the first node in a circular doubly linked list.

The node which is to be deleted can be the only node present in the linked list. In this case, the condition **head → next == head** will become true, therefore the list needs to be completely deleted.

In the second scenario, the list contains more than one element in the list, therefore the condition **head → next == head** will become false. Now, reach the last node of the list and make a few pointer adjustments there.

Now, temp will point to the node which is to be deleted from the list. Make the next pointer of previous node of temp, point to the head node of the list.

Make the previous pointer of the head node, point to the previous node of temp.

Now, free the temp pointer to free the memory taken by the node.

Algorithm

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 8

[END OF IF]

Step 2: SET TEMP = HEAD

Step 3: Repeat Step 4 while TEMP -> NEXT != HEAD

Step 4: SET TEMP = TEMP -> NEXT

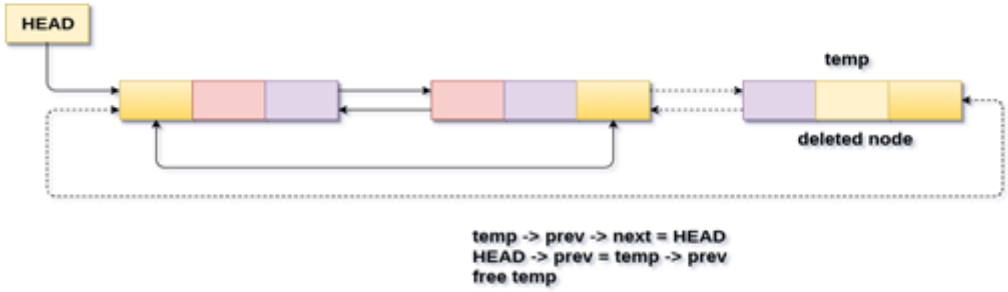
[END OF LOOP]

Step 5: SET TEMP -> PREV -> NEXT = HEAD

Step 6: SET HEAD -> PREV = TEMP -> PREV

Step 7: FREE TEMP

Step 8: EXIT



LINKED LIST IMPLEMENTATION OF STACK:

The major problem with the stack implemented using an array is, it works only for a fixed number of data values.

Stack implemented using an array is not suitable, when we don't know the size of data which we are going to use.

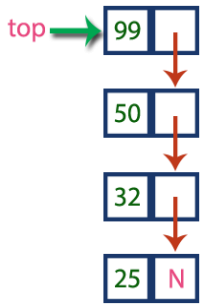
A stack data structure can be implemented by using a linked list data structure. The stack implemented using linked list can work for an unlimited number of values.

The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'.

Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list. The **next** field of the first element must be always **NULL**.

Example



Stack Operations using Linked List:

push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether stack is **Empty** (**top** == **NULL**)
- **Step 3 -** If it is **Empty**, then set **newNode → next** = **NULL**.
- **Step 4 -** If it is **Not Empty**, then set **newNode → next** = **top**.
- **Step 5 -** Finally, set **top** = **newNode**.

pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

- **Step 1 -** Check whether **stack** is **Empty** (**top == NULL**).
- **Step 2 -** If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function
- **Step 3 -** If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
- **Step 4 -** Then set '**top** = **top → next**'.
- **Step 5 -** Finally, delete '**temp**'. (**free(temp)**).

display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

- **Step 1 -** Check whether stack is **Empty** (**top** == **NULL**).
- **Step 2 -** If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty**, then define a Node pointer **'temp'** and initialize with **top**.
- **Step 4 -** Display '**temp → data** --->' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. (**temp → next** != **NULL**).
- **Step 5 -** Finally! Display '**temp → data** ---> **NULL**'.

#Python code to implement stack using linked list

```
class Node:

    # Class to create nodes of linked list
    # constructor initializes node automatically
    def __init__(self, data):
        self.data = data
        self.next = None


class Stack:

    # head is default NULL
    def __init__(self):
        self.head = None

    # Checks if stack is empty
    def isempty(self):
        if self.head == None:
            return True
        else:
            return False

    # Method to add data to the stack
    # adds to the start of the stack
    def push(self, data):

        if self.head == None:
            self.head = Node(data)

        else:
            newnode = Node(data)
            newnode.next = self.head
            self.head = newnode

    # Remove element that is the current head (start of the stack)
    def pop(self):

        if self.isempty():
            return None

        else:
            # Removes the head node and makes
            # the preceding one the new head
            poppednode = self.head
            self.head = self.head.next
```

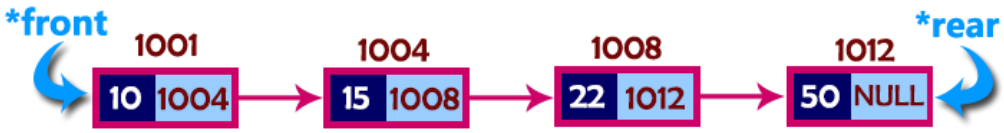
```
            poppednode.next = None
            return poppednode.data

    # Returns the head node data
    def peek(self):

        if self.isempty():
            return None

        else:
            return self.head.data

    # Prints out the stack
    def display(self):

        iternode = self.head
        if self.isempty():
            print("Stack Underflow")

        else:

            while (iternode != None):
                print(iternode.data, "->", end=" ")
                iternode = iternode.next
            return
if __name__ == "__main__":
    s = Stack()

    while(True):
        el = int(input("1 for Push\n2 for Pop\n3 to check if it is Empty\n4 to print
Stack\n5 to peek\n6 to exit\n"))
        if(el == 1):
            item = input("Enter Element to push in stack\n")
            s.push(item)
        if(el == 2):
            print(s.pop())
        if(el == 3):
            print(s.isEmpty())
        if(el == 4):
            s.printStack()
        if(el == 5):
            print(s.peek())
        if(el == 6):
            break
```

Queue implementation Using Linked List:

The major problem with the queue implemented using an array is, It will work for an only fixed number of data values. The queue which is implemented using a linked list can work for an unlimited number of values

In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.

Example



In above example, the last inserted node is 50 and it is pointed by '**rear**' and the first inserted node is 10 and it is pointed by '**front**'. The order of elements inserted is 10, 15, 22 and 50.

Operations:

enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue...

- **Step 1 -** Create a **newNode** with given value and set '**newNode → next**' to **NULL**.
- **Step 2 -** Check whether queue is **Empty** (**rear** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **front** = **newNode** and **rear** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, set **rear →** **next** = **newNode** and **rear** = **newNode**.

deQueue() - Deleting an Element from Queue

We can use the following steps to delete a node from the queue...

- **Step 1 -** Check whether **queue** is **Empty** (**front == NULL**).
- **Step 2 -** If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** and terminate from the function
- **Step 3 -** If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.
- **Step 4 -** Then set '**front** = **front → next**' and delete '**temp**' (**free(temp)**).

display() - Displaying the elements of Queue

We can use the following steps to display the elements (nodes) of a queue...

- **Step 1 -** Check whether queue is **Empty** (**front** == **NULL**).
- **Step 2 -** If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **front**.
- **Step 4 -** Display '**temp → data** --->' and move it to the next node. Repeat the same until '**temp**' reaches to '**rear**' (**temp → next** != **NULL**).
- **Step 5 -** Finally! Display '**temp → data** ---> **NULL**'.

#python code to implement queue using linked list

```
class Node:
    def __init__(self,data):
        self.data = data
        self.next = None
class Queue:
    def __init__(self):
        self.head = None
    def display(self):
        temp = self.head
        print('Queue is:[', end='')
        while temp:
            print(temp.data, end=", ")
            temp = temp.next
        print(']')
    def enqueue(self, data):
        if self.head is None:
            self.head = Node(data)
        else:
```

```
            temp = self.head
            while temp.next is not None:
                temp = temp.next
            temp.next = Node(data)
        #self.display()
    def dequeue(self):
        if self.head is None:
            print('Queue is Empty')
            #self.display()
            #return None
        else:
            data=self.head.data
            self.head=self.head.next
            #self.display()
            return data
if __name__ == '__main__':
    Q = Queue()
    print('1.enqueue\n2.dequeue\n3.display\n4.exit')
    while 1:
        choice = input('Enter your option:')
        if choice is '1':
            d = int(input('Enter data:'))
            Q.enqueue(d)
        elif choice is '2':
            print('poped element is:',Q.dequeue())
        elif choice is '3':
            print(Q.display())
        else:
            break
```