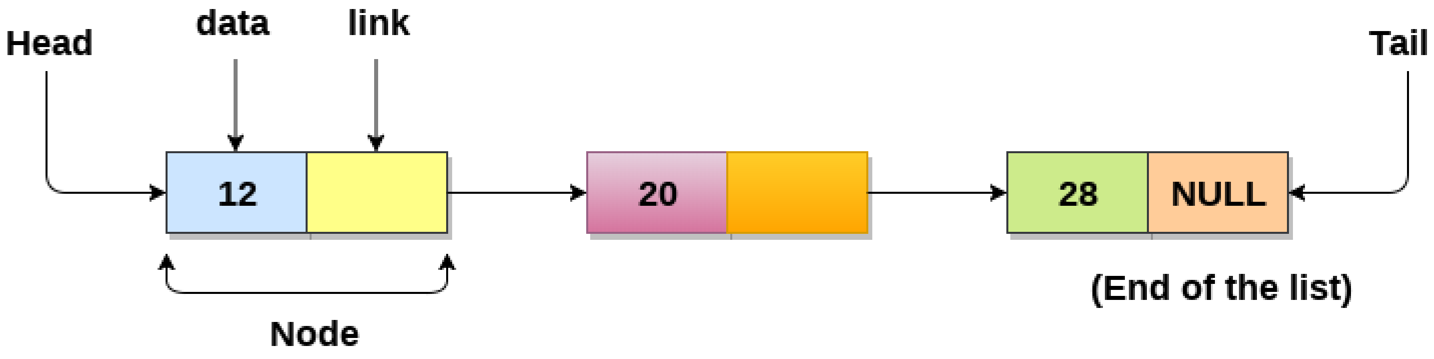
Unit – 2
Linked List Data Structure

What is a Linked List?

- Linked List can be defined as a collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains a pointer to the null.



Uses of Linked List

- The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node can not be present in the linked list.
- We can store values of primitive types or objects in the singly linked list.

Why use linked list over array?

Till now, we were using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages which must be known in order to decide the data structure which will be used throughout the program.
Array contains following limitations:

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

Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,

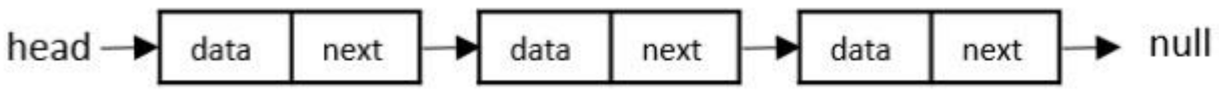
1. It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
2. Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and is limited to the available memory space.

Types of Linked List

Following are the various types of linked list.

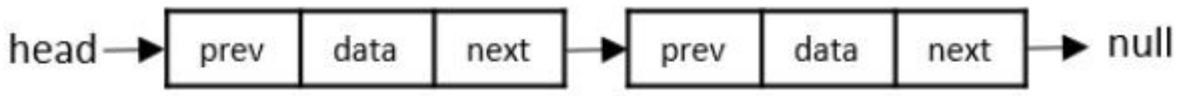
Singly Linked Lists

Singly-linked lists contain two "buckets" in one node; one bucket holds the data and the other bucket holds the address of the next node of the list. Traversals can be done in one direction only as there is only a single link between two nodes of the same list.



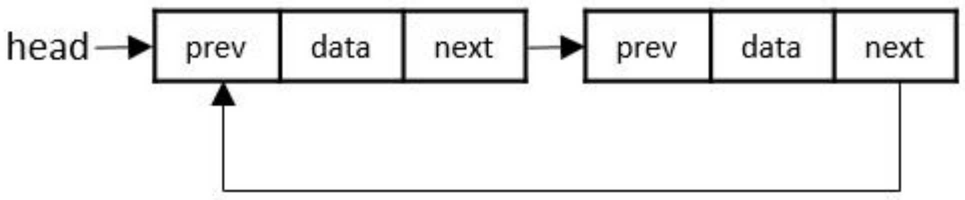
Doubly Linked Lists

Doubly Linked Lists contain three "buckets" in one node; one bucket holds the data and the other buckets hold the addresses of the previous and next nodes in the list. The list is traversed twice as the nodes in the list are connected from both sides.



Circular Linked Lists

Circular linked lists can exist in both singly linked list and doubly linked list.
Since the last node and the first node of the circular linked list are connected, the traversal in this linked list will go on forever until it is broken.



Basic Operations in Linked List

The basic operations in the linked lists are insertion, deletion, searching, display, and deleting an element at a given key. These operations are performed on Singly Linked Lists as given below −

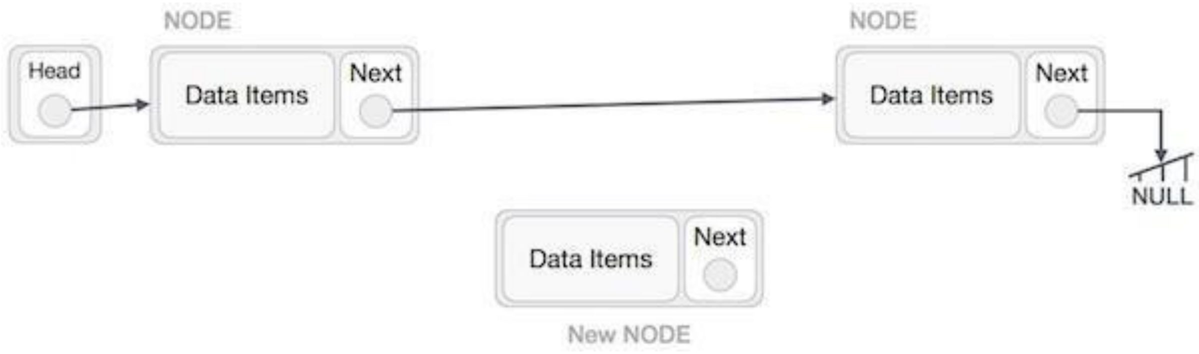
- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Display** − Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

Node Creation

```
struct node
{
   int data;
   struct node *next;
};
struct node *head, *ptr;
ptr = (struct node *)malloc(sizeof(struct node *));
```

Linked List - Insertion Operation

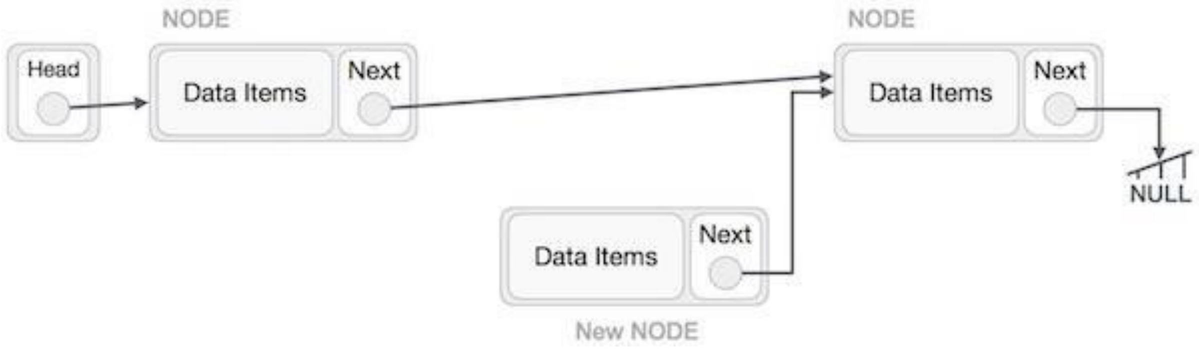
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node B (NewNode), between A (LeftNode) and C (RightNode). Then point B.next to C −

NewNode.next -> RightNode;

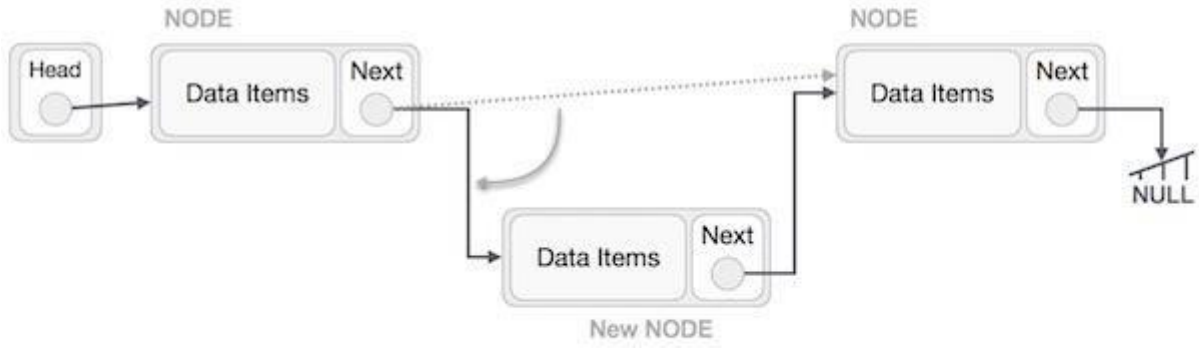
It should look like this −



Now, the next node at the left should point to the new node.

LeftNode.next -> NewNode;

This will put the new node in the middle of the two. The new list should look like this −



Insertion in linked list can be done in three different ways. They are explained as follows −

Insertion at Beginning

Inserting a new element into a singly linked list at the beginning is quite simple. We just need to make a few adjustments in the node links. There are the following steps that need to be followed to insert a new node in the list at the beginning.

- Allocate the space for the new node and store data in the data part of the node. This will be done by the following statements.

  *ptr = (struct node *) malloc(sizeof(struct node *));*

  ptr → data = item

- Make the link part of the new node pointing to the existing first node of the list. This will be done by using the following statement.

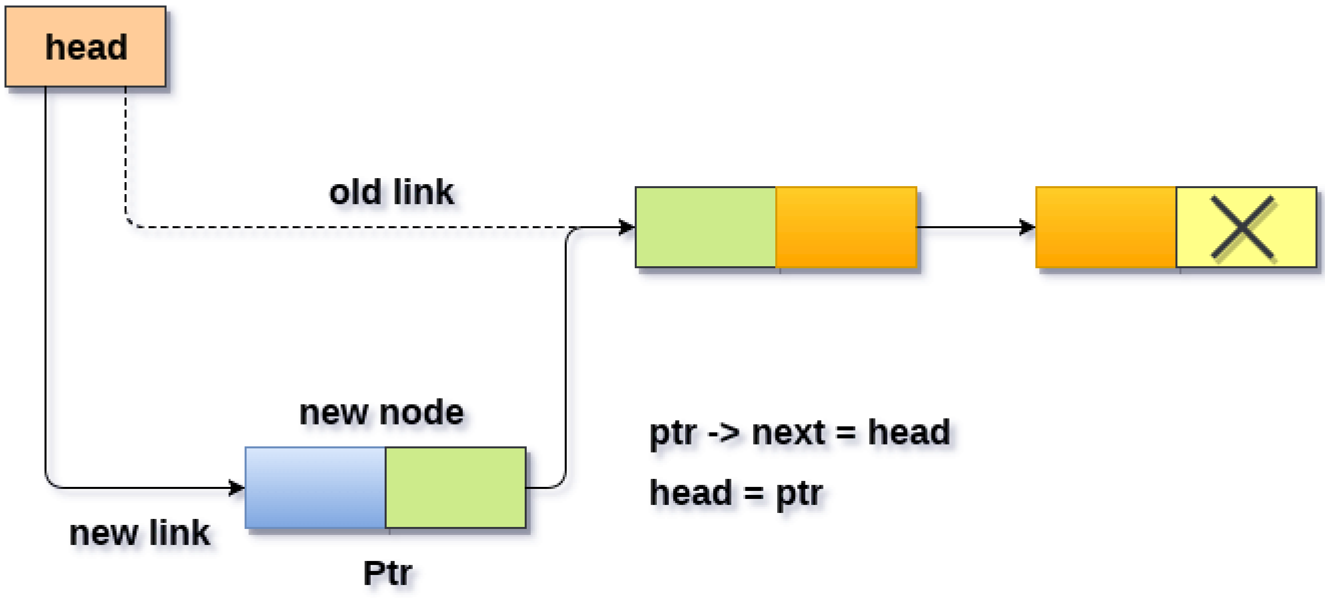
  ptr->next = head;

- 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.

  head = ptr;

Algorithm

```
o Step 1: IF NEW_NODE = NULL
  Write OVERFLOW
      Go to Step 7
     [END OF IF]
o Step 2: SET NEW_NODE → DATA = VAL
o Step 3: SET NEW_NODE → NEXT = HEAD
o Step 4: SET HEAD = NEW_NODE
o Step 5: EXIT
```



Insertion at Ending

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 by using malloc statement in C. Data and the link part of the node are set up by using the following statements.

  ```
  ptr->data = item;
  ptr -> next = NULL;
  ```

- 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. This will be done by using the following Statements.

  ```
  Head = ptr
  ```

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.

  ```
  Temp = head
  ```

- Then, traverse through the entire linked list using the statements:

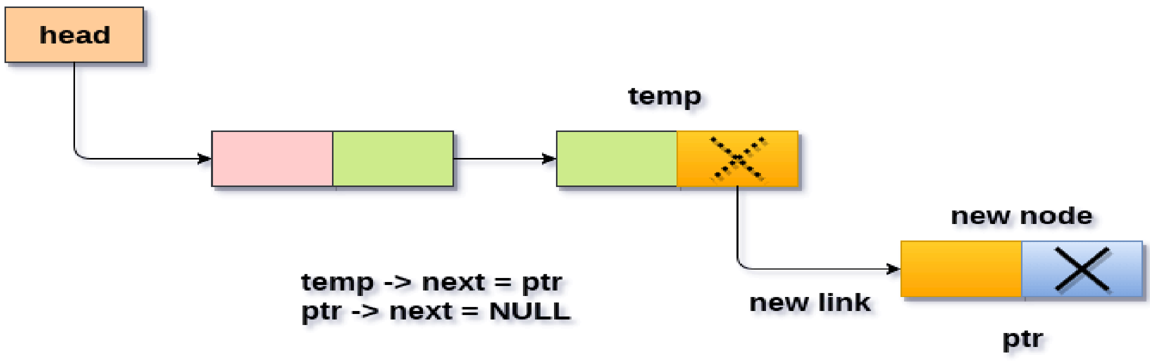
  ```
  while (temp→ next != NULL)
    temp = temp → next;
  ```

- 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) .

```
temp = head;
  while (temp -> next != NULL)
  {
     temp = temp -> next;
  }
  temp->next = ptr;
  ptr->next = NULL;
```

Algorithm

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



Inserting node at the last into a non-empty list

<u>Insertion at a Given Position</u>

- 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. This will be done by using the following statements.
  emp=head;

```
for(i=0;i<loc;i++)
{
    temp = temp->next;
    if(temp == NULL)
    {
        return;
    }

}
```

- Allocate the space for the new node and add the item to the data part of it. This will be done by using the following statements.

```
ptr = (struct node *) malloc (sizeof(struct node));
 ptr->data = item;
```

- 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). This will be done by using the following statements.

  ptr→ next = temp → next

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.

temp ->next = ptr;

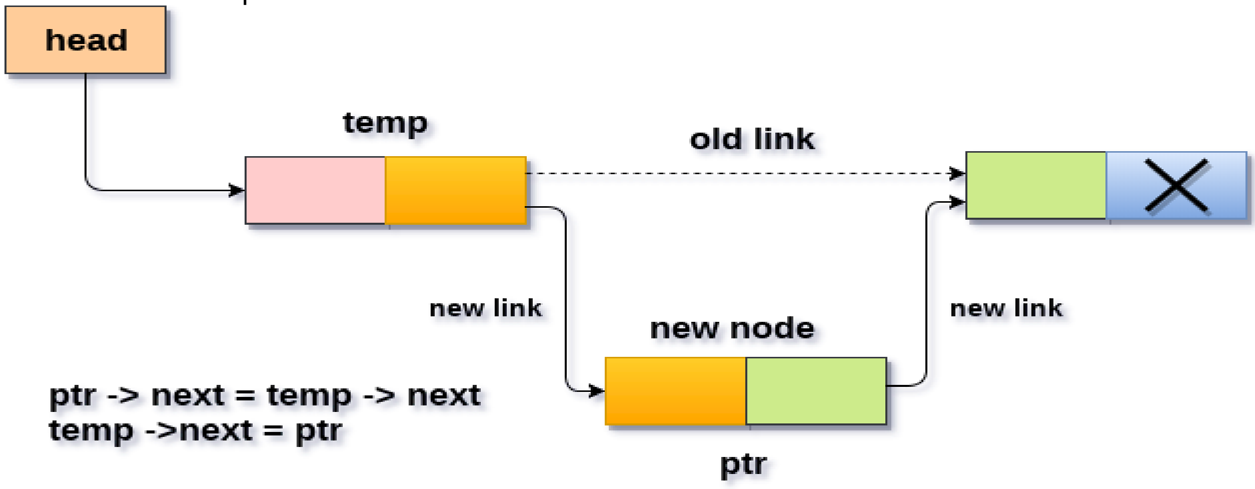
Algorithm

- Step 1: Declaration
- Step 2: IF ptr = NULL Write OVERFLOW
- Go to Step 1
- [END OF IF]
- Step 3: Create a new node and SET ptr - > DATA = VAL, read the loc.
- Step 4: SET temp=head;

```
for(i=1;i<loc;i++)
{
        temp = temp->next;
        if(temp == NULL)
        {
        printf("\ncan't insert\n");
        return;
        }

}
```

- ptr ->next = temp ->next;
- temp ->next = ptr;
- Step 4:printf "Node inserted"

- Step 5:Exit



Linked List - Deletion Operation

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.

Deletion in singly linked list at beginning

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. This will be done by using the following statements.

ptr = head;

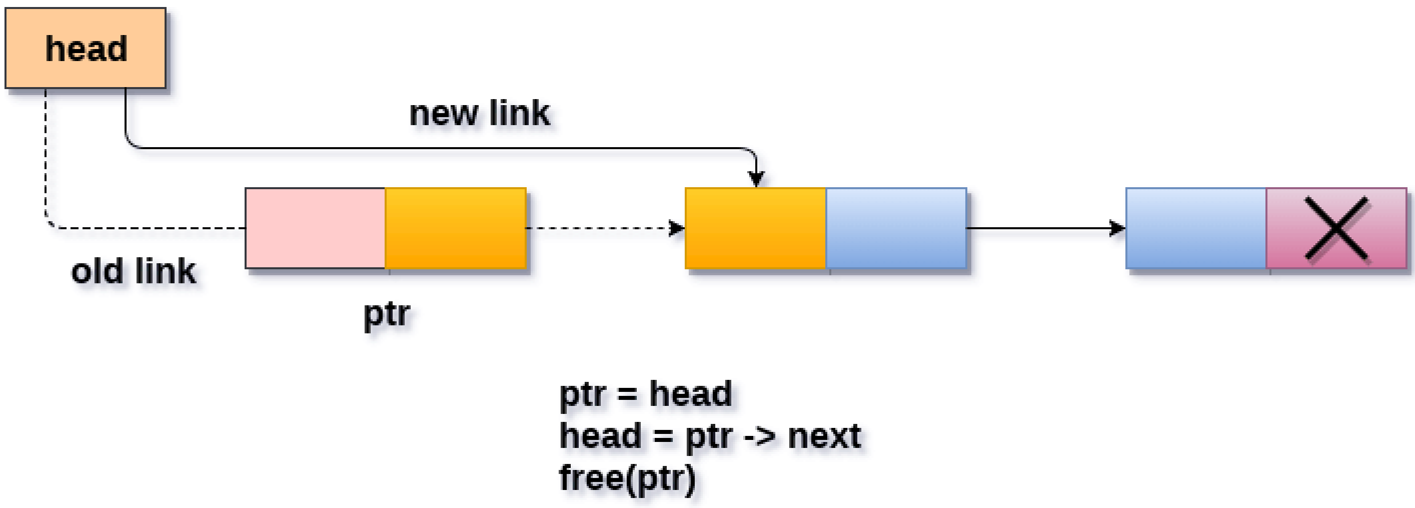
head = ptr->next;

Now, free the pointer ptr which was pointing to the head node of the list. This will be done by using the following statement.

free(ptr)

Algorithm

```
o Step 1: IF HEAD = NULL
  Write UNDERFLOW
      Go to Step 5
     [END OF IF]
o Step 2: SET PTR = HEAD
o Step 3: SET HEAD = HEAD -> NEXT
o Step 4: FREE PTR
o 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. This will be done by using the following statements.

```
ptr = head
head = NULL
free(ptr)
```

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.

this all will be done by using the following statements.

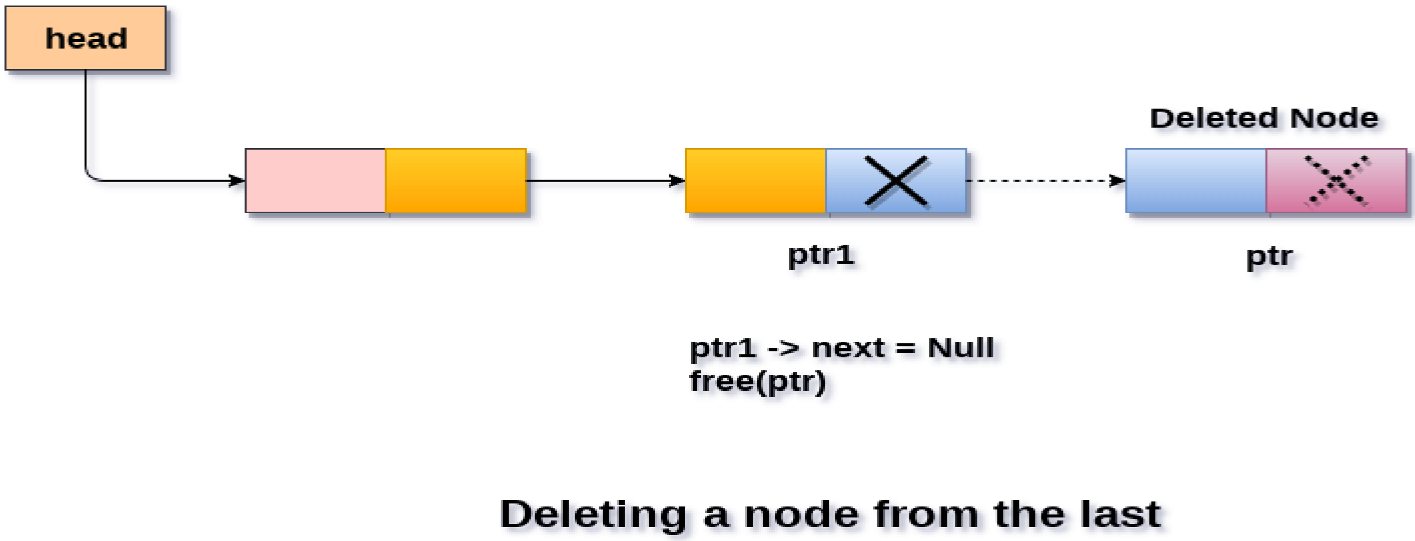
```
ptr = head;
while(ptr->next != NULL)
{
    ptr1 = ptr;
    ptr = ptr ->next;
}
```

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. It will be done by using the following statements.

```
ptr1->next = NULL;
free(ptr);
```

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
```



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.
Use the following statements to do so.

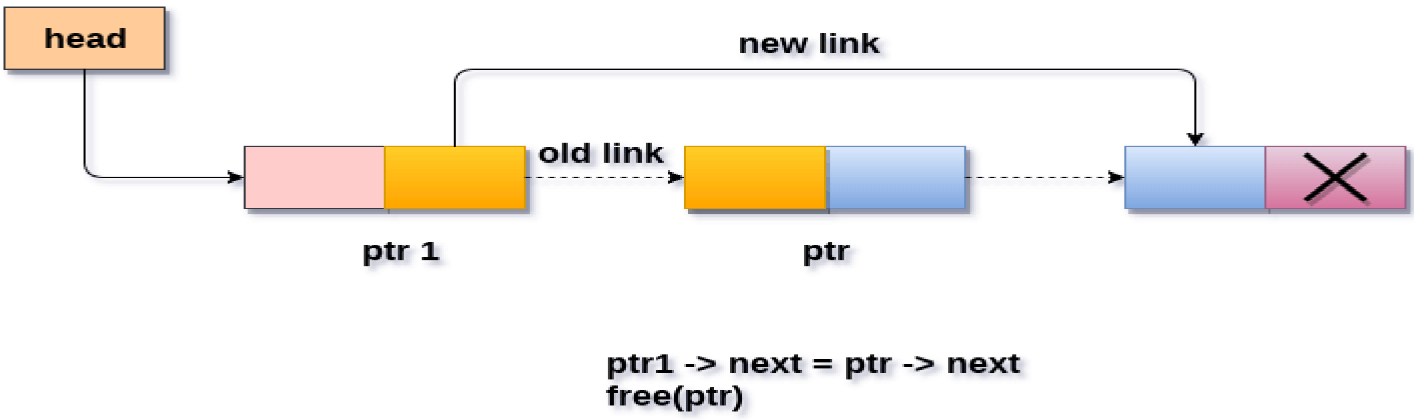
```
ptr=head;
  for(i=0;i<loc;i++)
  {
     ptr1 = ptr;
     ptr = ptr->next;

     if(ptr == NULL)
     {
        printf("\nThere are less than %d elements in the list..",loc);
        return;
     }
}
```

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).
This will be done by using the following statements.

```
ptr1 ->next = ptr ->next;
 free(ptr);
```

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
○ 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.

```
ptr = head;
 while (ptr!=NULL)
```

```
{
   ptr = ptr -> next;
}
```

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

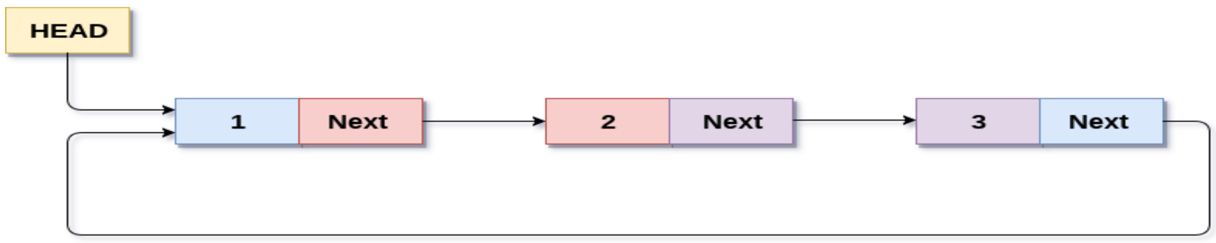
```
Step 1: SET PTR = HEAD
Step 2: Set I = 0
STEP 3: IF PTR = NULL
          WRITE "EMPTY LIST"
          GOTO STEP 8
          END OF IF
STEP 4: REPEAT STEP 5 TO 7 UNTIL PTR != NULL
STEP 5: if ptr → data = item
          write i+1
         End of IF
STEP 6: I = I + 1
STEP 7: PTR = PTR → NEXT
        [END OF LOOP]
STEP 8: EXIT
```

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.



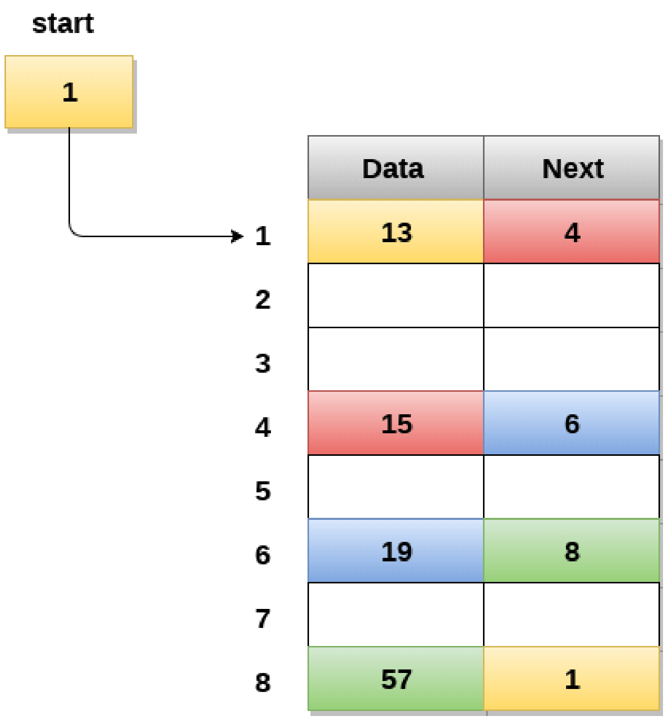
Circular Singly Linked List

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, the image shows a glimpse of how the circular list is being stored in the memory. The start or head of the list is pointing to the element with the index 1 and containing 13 marks in the data part and 4 in the next part. Which means that it is linked with the node that is being stored at 4th index of the list.

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.



	Data	Next
1	13	4
2		
3		
4	15	6
5		
6	19	8
7		
8	57	1

Memory Representation of a circular linked list

We can also have more than one number of linked list in the memory with the different start pointers pointing to the different start nodes in the list. The last node is identified by its next part which contains the address of the start node of the list. We must be able to identify the last node of any linked list so that we can find out the number of iterations which need to be performed while traversing the 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 scenario 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. Firstly, allocate the memory space for the new node by using the malloc method of C language.

```
struct node *ptr = (struct node *)malloc(sizeof(struct node));
```

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. This will be done by using the following statements.

```
if(head == NULL)
    {
        head = ptr;
        ptr -> next = head;
    }
```

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.

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

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.
This will be done by using the following statements.

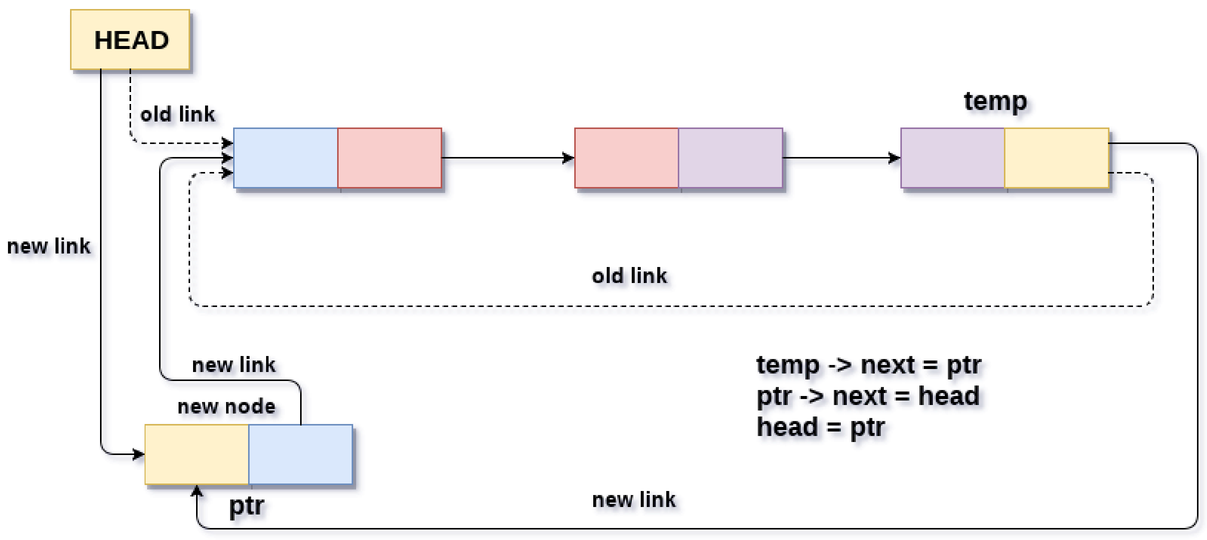
temp -> next = ptr;
the next pointer of temp will point to the existing head node of the list.

ptr->next = head;
Now, make the new node ptr, the new head node of the circular singly linked list.

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

Algorithm

```
o   Step 1: IF PTR = NULL
      Write OVERFLOW
     Go to Step 11
     [END OF IF]
o   Step 2: Create a new node  PTR
o   Step 3: SET ptr -> data = VAL
o   Step 4: SET temp = head
o   Step 5: Repeat Step 8 while temp -> next != head
o   Step 6: SET temp = temp -> next
    [END OF LOOP]
o   Step 7: SET ptr -> next = head
o   Step 8: SET temp → next = ptr
o   Step 9: SET head = ptr
o   Step 10: EXIT
```



Insertion into circular singly linked list at beginning

Insertion into circular singly linked list at the end

There are two scenario 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.

- Firstly, allocate the memory space for the new node by using the malloc method of C language.

```
struct node *ptr = (struct node *)malloc(sizeof(struct node));
```

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. This will be done by using the following statements.

```
if(head == NULL)
    {
       head = ptr;
       ptr -> next = head;
    }
```

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.

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

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**. This is done by using the following statement.

```
temp -> next = ptr;
```

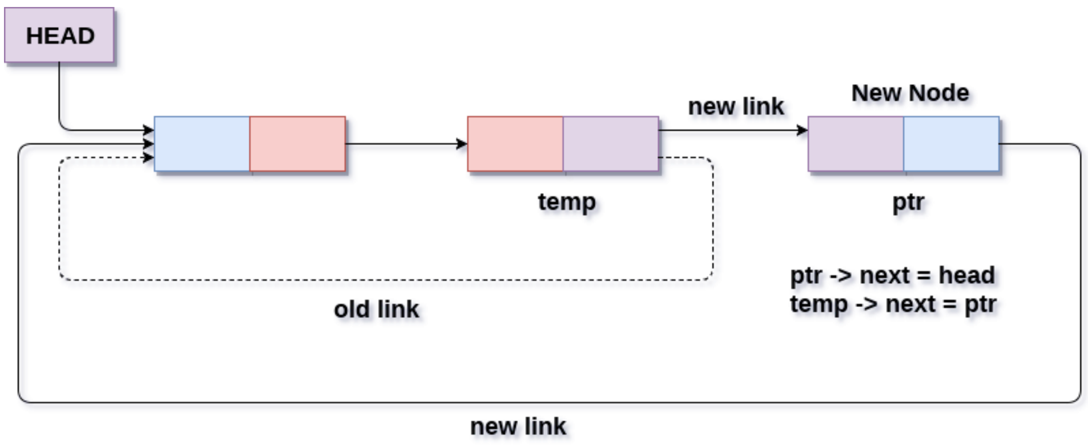
The new last node of the list i.e. ptr will point to the head node of the list.

```
ptr -> next = head;
```

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

Algorithm

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

```
if(head == NULL)
  {
    printf("\nUNDERFLOW");
    return;
  }
```

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. This will be done by using the following statements.

```
if(head->next == head)
{
  head = NULL;
  free(head);
}
```

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. This will be done by using the following statements.

```
ptr = head;
while(ptr -> next != head)
  ptr = ptr -> next;
```

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.

```
ptr->next = head->next;
```

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

```
free(head);
```

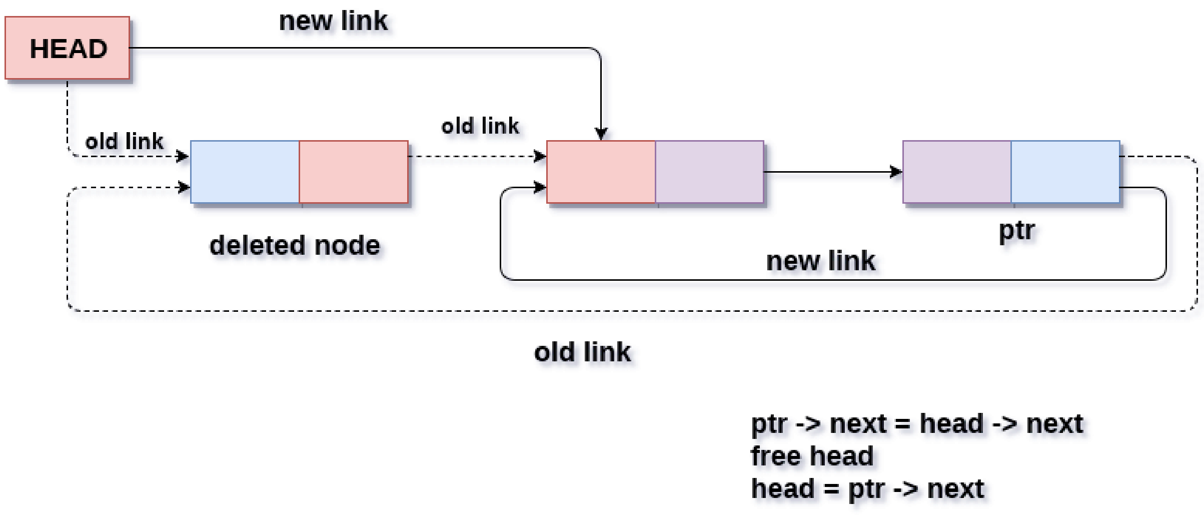
Make the node pointed by the next of the last node, the new head of the list.

```
head = ptr->next;
```

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

Algorithm

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

```
if(head == NULL)
  {
    printf("\nUNDERFLOW");
    return;
  }
```

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. This will be done by using the following statements.

```
if(head->next == head)
{
  head = NULL;
  free(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. The following sequence of code is used for this purpose.

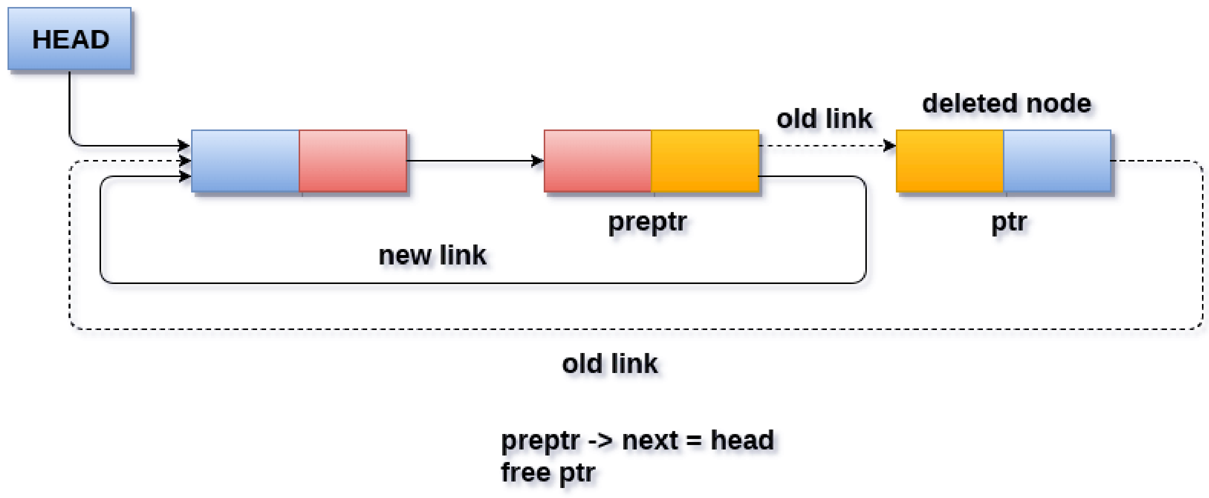
```
ptr = head;
    while(ptr ->next != head)
    {
      preptr=ptr;
      ptr = ptr->next;
    }
```

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.

```
preptr->next = ptr -> next;
 free(ptr);
```

Algorithm

```
o  Step 1: IF HEAD = NULL
     Write UNDERFLOW
      Go to Step 8
     [END OF IF]
o  Step 2: SET PTR = HEAD
o  Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != HEAD
o  Step 4: SET PREPTR = PTR
o  Step 5: SET PTR = PTR -> NEXT
   [END OF LOOP]
o  Step 6: SET PREPTR -> NEXT = HEAD
o  Step 7: FREE PTR
o  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.
The algorithm and its implementation in C is given as follows.

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: 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

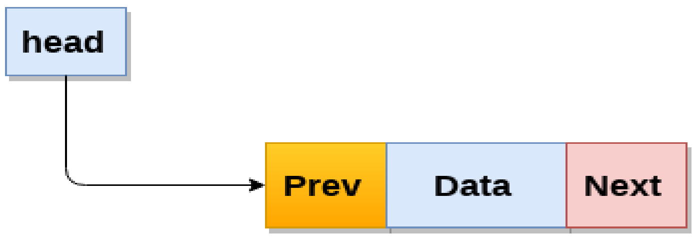
Traversing in circular singly linked list can be done through a loop. Initialize the temporary pointer variable **temp** to head pointer and run the while loop until the next pointer of temp becomes **head**. The algorithm and the c function implementing the algorithm is described as follows.

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
```

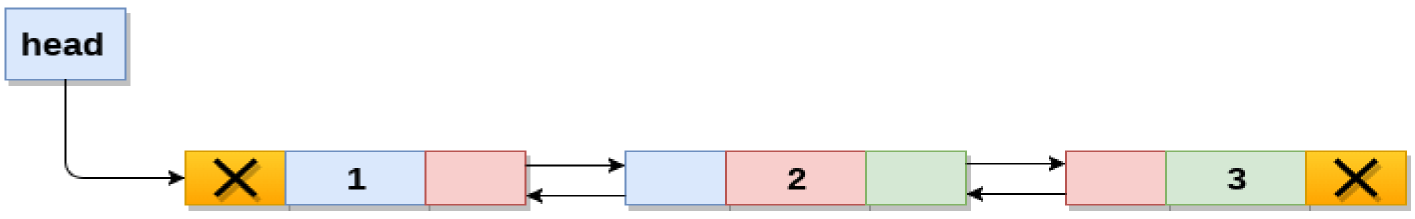
Doubly linked list

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

In C, structure of a node in doubly linked list can be given as :

```
struct node
{
   struct node *prev;
```

```
    int data;
    struct node *next;
}
```

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

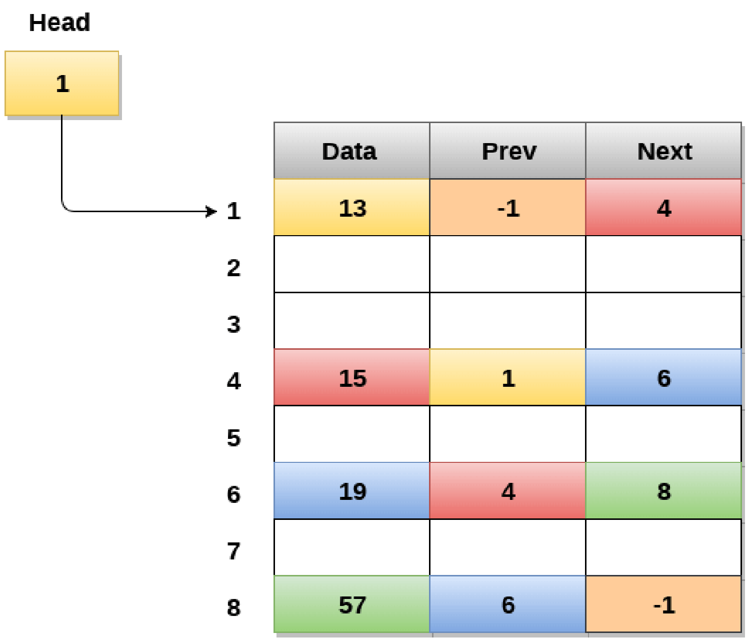
In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains 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).

In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev** of the list **contains** null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

We can traverse the list in this way until we find any node containing null or -1 in its next part.



	Data	Prev	Next
1	13	-1	4
2			
3			
4	15	1	6
5			
6	19	4	8
7			
8	57	6	-1

Memory Representation of a Doubly linked list

Operations on doubly linked list

Node Creation

```
struct node
{
    struct node *prev;
    int data;
    struct node *next;
```

```
        };
        struct node *head;
```

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. This will be done by using the following statement.

```
ptr = (struct node *)malloc(sizeof(struct node));
```

- 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.

```
ptr->next = NULL;
    ptr->prev=NULL;
   ptr->data=item;
    head=ptr;
```

- 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.
- This will be done by using the following statements.

ptr->next = head;

head→prev=ptr;

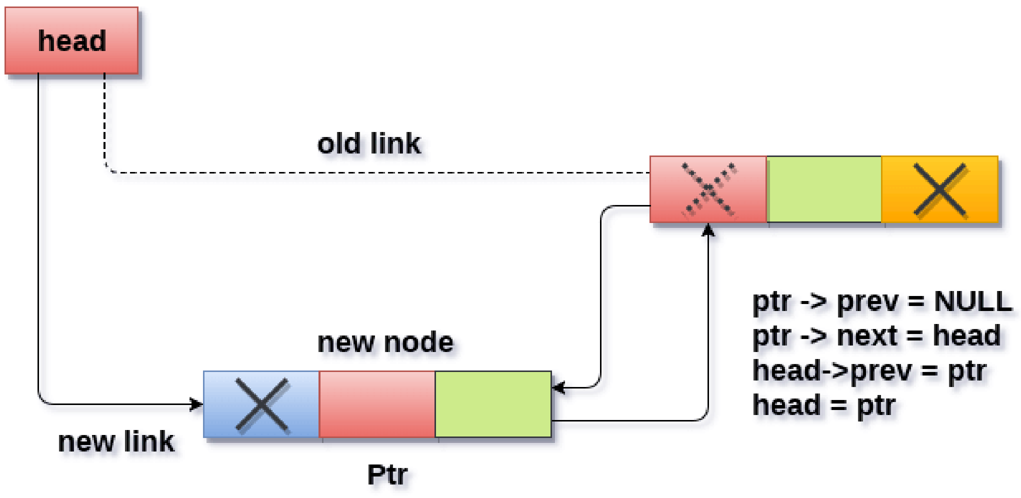
Since, the node being inserted is the first node of the list and therefore it must contain NULL in its prev pointer. Hence assign null to its previous part and make the head point to this node.

ptr→prev =NULL

head = ptr

Algorithm :

```
o  Step 1: IF ptr = NULL
     Write OVERFLOW
    Go to Step 9
    [END OF IF]
o  Step 2: Create a NEW_NODE
o  Step 3: SET NEW_NODE -> DATA = VAL
o  Step 4: SET NEW_NODE -> PREV = NULL
o  Step 5: SET NEW_NODE -> NEXT = START
o  Step 6: SET head -> PREV = NEW_NODE
o  Step 7: SET head = NEW_NODE
o  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.

ptr = (struct node *) malloc(sizeof(struct node));

- 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.

ptr->next = NULL;

```
ptr->prev=NULL;
ptr->data=item;
head=ptr;
```

- 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.

```
Temp = head;
    while (temp != NULL)
    {
    temp = temp → next;
      }
```

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.

```
temp→next =ptr;
```

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

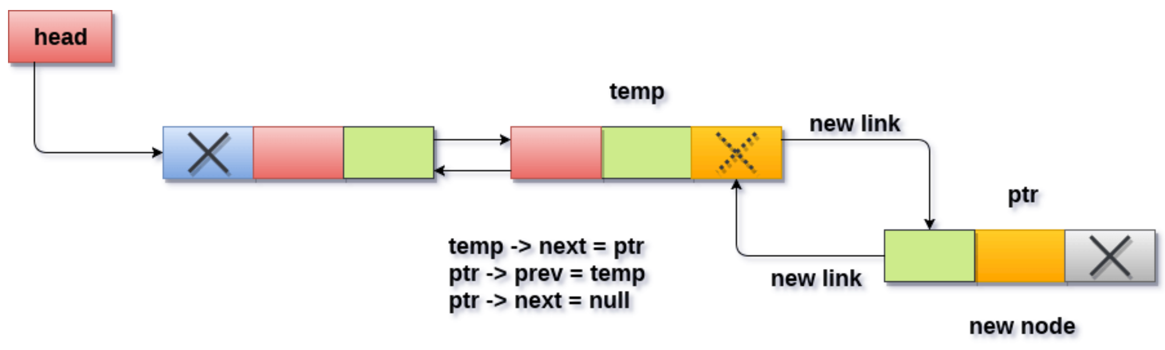
```
ptr → prev = temp;
```

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

```
ptr → next = NULL
```

Algorithm

```
o Step 1: IF PTR = NULL
      Write OVERFLOW
       Go to Step 11
      [END OF IF]
o Step 2: Create a NEW_NODE
o Step 3: SET NEW_NODE -> DATA = VAL
o Step 4: SET NEW_NODE -> NEXT = NULL
o Step 5: SET TEMP = START
o Step 6: Repeat Step 8 while TEMP -> NEXT != NULL
o Step 7: SET TEMP = TEMP -> NEXT
     [END OF LOOP]
o Step 8: SET TEMP -> NEXT = NEW_NODE
o Step 9 : SET NEW_NODE -> PREV = TEMP
o 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. Use the following statements for this.

```
ptr = (struct node *)malloc(sizeof(struct node));
```

Traverse the list by using the pointer **temp** to skip the required number of nodes in order to reach the specified node.

```
temp=head;
 for(i=0;i<loc;i++)
 {
   temp = temp->next;
   if(temp == NULL) // the temp will be //null if the list doesn't last long //up to mentioned location
   {
      return;
   }
 }
```

- 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 fer pointer adjustments here. Make the next pointer of **ptr** point to the next node of temp.

```
ptr → next = temp → next;
```

make the **prev** of the new node ptr point to temp.

```
ptr → prev = temp;
```

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

```
temp → next = ptr;
```

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

```
temp → next → prev = ptr;
```

Algorithm

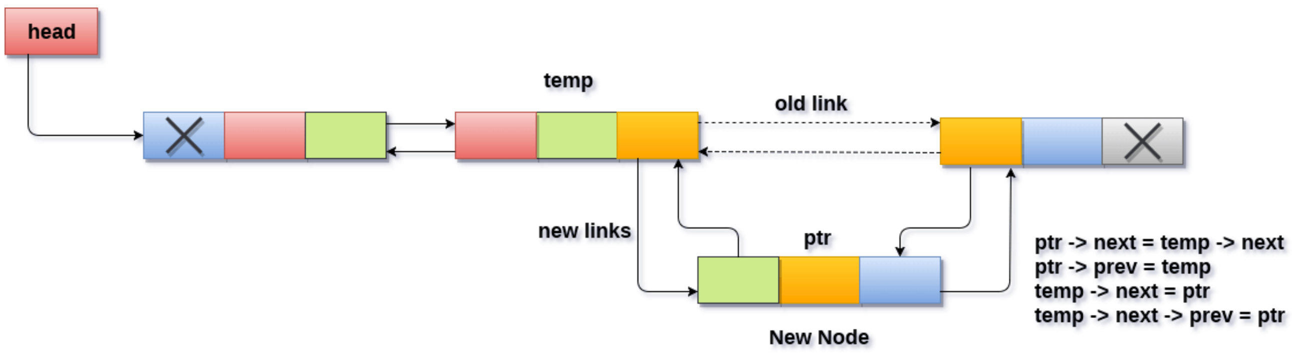
- **Step 1:** IF PTR = NULL

```
            Write OVERFLOW
            Go to Step 15
          [END OF IF]
```

- **Step 2:** SET NEW_NODE = PTR
- **Step 3:** SET PTR = PTR -> NEXT
- **Step 4:** SET NEW_NODE -> DATA = VAL
- **Step 5:** SET TEMP = START
- **Step 6:** SET I = 0
- **Step 7:** REPEAT 8 to 10 until I
- **Step 8:** SET TEMP = TEMP -> NEXT
- **STEP 9:** IF TEMP = NULL
- **STEP 10:** WRITE "LESS THAN DESIRED NO. OF ELEMENTS"

```
            GOTO STEP 15
            [END OF IF]
          [END OF LOOP]
```

- **Step 11:** SET NEW_NODE -> NEXT = TEMP -> NEXT
- **Step 12:** SET NEW_NODE -> PREV = TEMP
- **Step 13 :** SET TEMP -> NEXT = NEW_NODE
- **Step 14:** SET TEMP -> NEXT -> PREV = NEW_NODE
- **Step 15:** 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.

Ptr = head;

head = head → next;

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

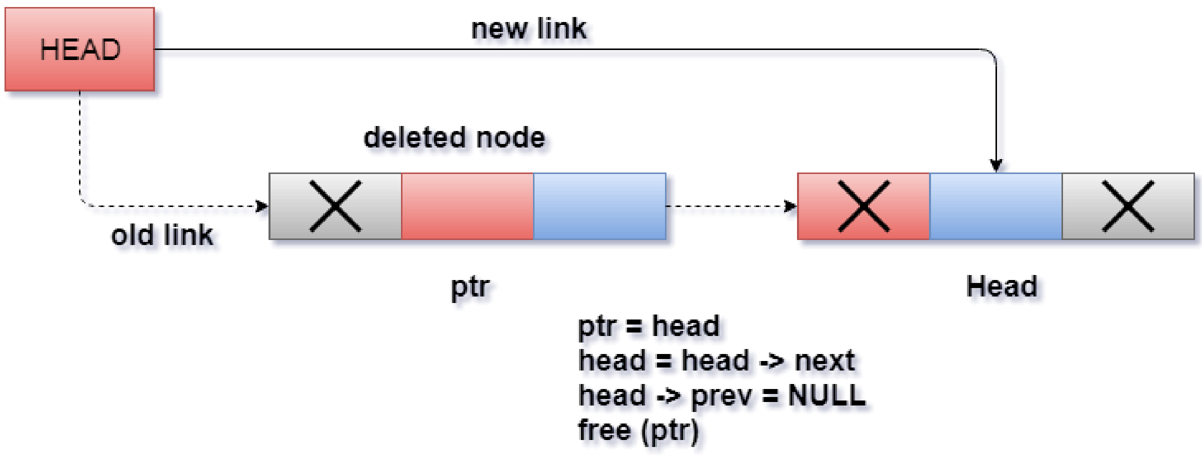
head → prev = NULL

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

free(ptr)

Algorithm

- **STEP 1:** IF HEAD = NULL
  WRITE UNDERFLOW
  GOTO STEP 6
- **STEP 2:** SET PTR = HEAD
- **STEP 3:** SET HEAD = HEAD → 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.

```
ptr = head;
   if(ptr->next != NULL)
   {
      ptr = ptr -> next;
   }
```

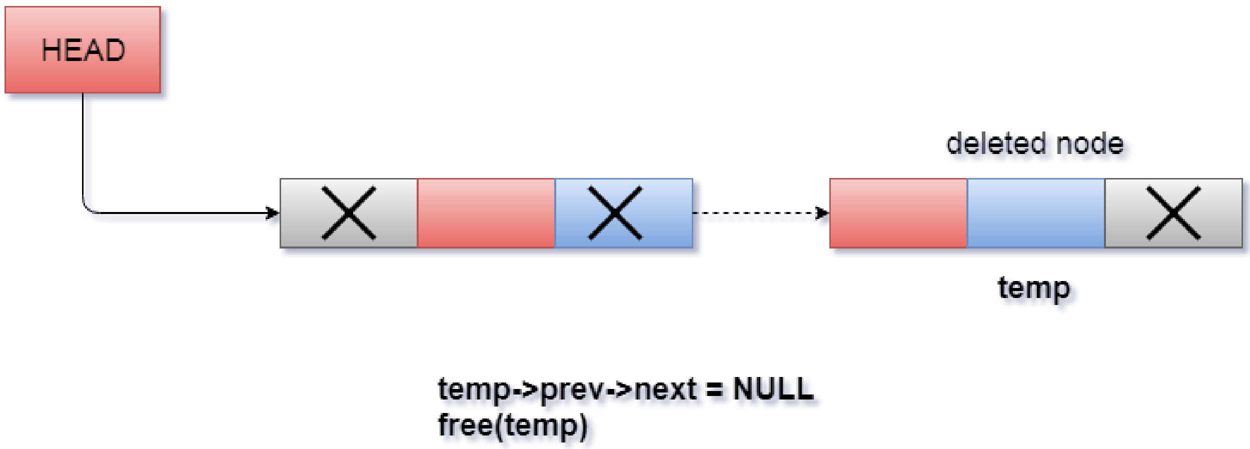
- The ptr would point to the last node of the ist at the end of the for loop. Just make the next pointer of the previous node of **ptr** to **NULL**.

ptr → prev → next = NULL

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

free(ptr)

```
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.

temp = head

- Traverse the list until we find the desired data value.

```
while(temp -> data != val)
temp = temp -> next;
```

- Check if this is the last node of the list. If it is so then we can't perform deletion.

```
if(temp -> next == NULL)
  {
return;
  }
```

- 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.

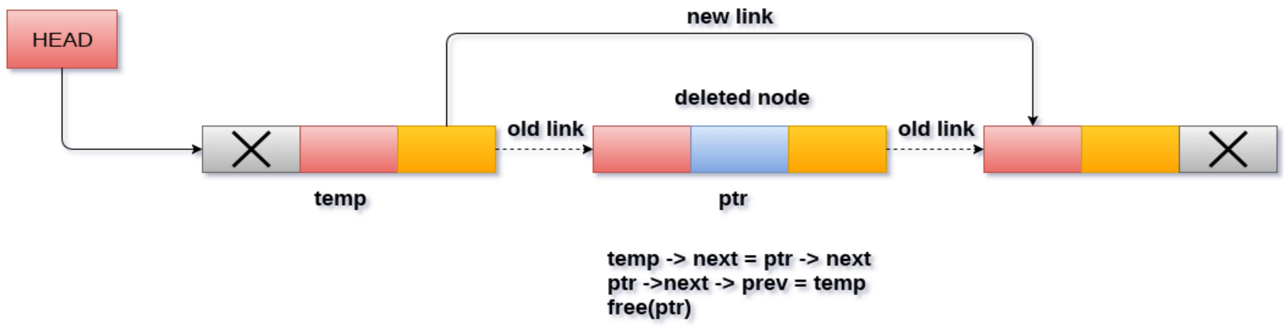
```
if(temp -> next -> next == NULL)
  {
     temp ->next = NULL;
  }
```

- 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.

```
ptr = temp -> next;
     temp -> next = ptr -> next;
     ptr -> next -> prev = temp;
     free(ptr);
```

Algorithm

```
o Step 1: IF HEAD = NULL
     Write UNDERFLOW
    Go to Step 9
   [END OF IF]
o Step 2: SET TEMP = HEAD
o Step 3: Repeat Step 4 while TEMP -> DATA != ITEM
o Step 4: SET TEMP = TEMP -> NEXT
    [END OF LOOP]
o Step 5: SET PTR = TEMP -> NEXT
o Step 6: SET TEMP -> NEXT = PTR -> NEXT
o Step 7: SET PTR -> NEXT -> PREV = TEMP
o Step 8: FREE PTR
o Step 9: EXIT
```



Deletion of a specified node in doubly linked list

<u>Searching for a specific node in Doubly Linked List</u>

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.

ptr = head

- declare a local variable I and assign it to 0.

i=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

```
o Step 1: IF HEAD == NULL
    WRITE "UNDERFLOW"
   GOTO STEP 8
   [END OF IF]
o Step 2: Set PTR = HEAD
o Step 3: Set i = 0
o Step 4: Repeat step 5 to 7 while PTR != NULL
o Step 5: IF PTR → data = item
  return i
  [END OF IF]
o Step 6: i = i + 1
o Step 7: PTR = PTR → next
o 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.

Ptr = head

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.

```
while(ptr != NULL)
   {
      printf("%d\n",ptr->data);
      ptr=ptr->next;
   }
```

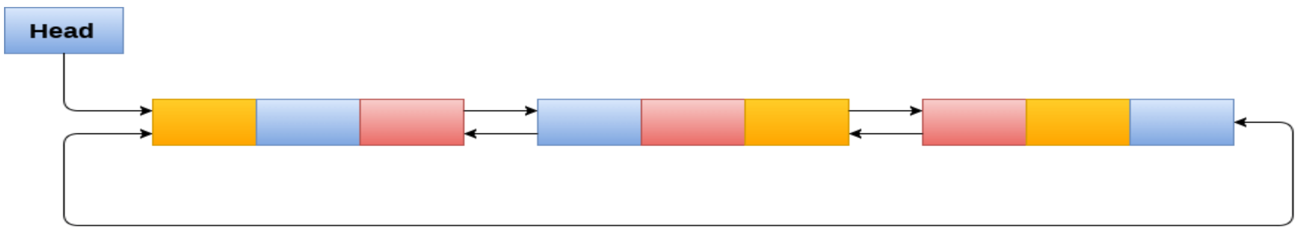
Although, traversing means visiting each node of the list once to perform some specific operation. Here, we are printing the data associated with each node of the list.

Algorithm

```
o Step 1: IF HEAD == NULL
    WRITE "UNDERFLOW"
   GOTO STEP 6
   [END OF IF]
o Step 2: Set PTR = HEAD
o Step 3: Repeat step 4 and 5 while PTR != NULL
o Step 4: Write PTR → data
o Step 5: PTR = PTR → next
o Step 6: Exit
```

Circular Doubly Linked List

Circular doubly linked list is a more complexed type of data structure in which a node contain 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 contain address of the last node in its previous pointer.
A circular doubly linked list is shown in the following figure.

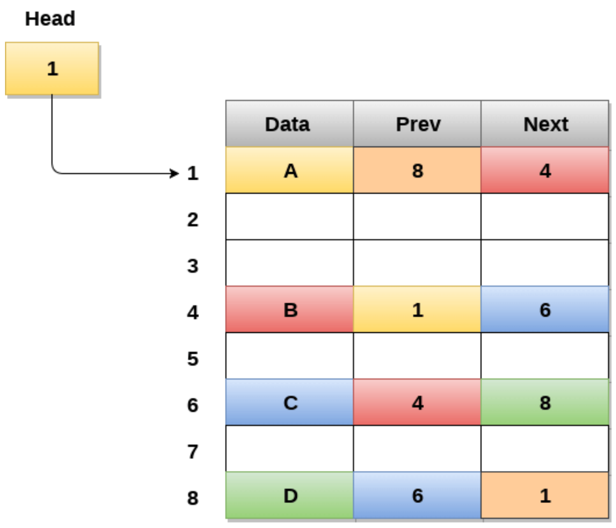


Circular Doubly Linked List

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 i.e. 1. 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.



	Data	Prev	Next
1	A	8	4
2			
3			
4	B	1	6
5			
6	C	4	8
7			
8	D	6	1

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 is 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.

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.

```
ptr = (struct node *)malloc(sizeof(struct node));
```

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. This can be done by using the following statement.

```
head = ptr;
    ptr -> next = head;
    ptr -> prev = head;
```

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. Traversing the list can be done by using the following statements.

```
temp = head;
while(temp -> next != head)
{
   temp = temp -> next;
}
```

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. All the pointer adjustments can be done by using the following statements.

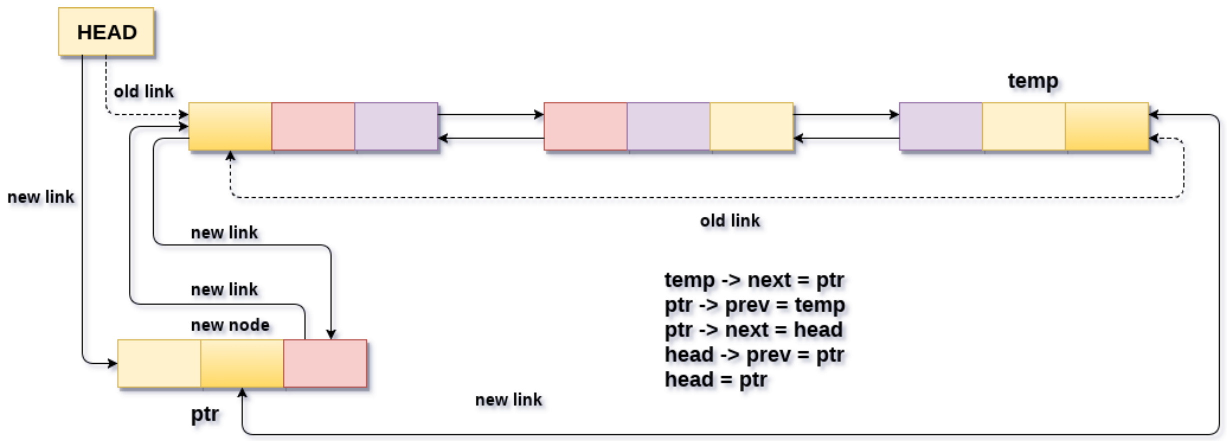
```
temp -> next = ptr;
   ptr -> prev = temp;
   head -> prev = ptr;
   ptr -> next = head;
   head = ptr;
```

In this way, the new node is inserted into the list at the beginning. The algorithm and its C implementation is given as follows.

Algorithm

- **Step 1:** IF PTR = NULL

```
        Write OVERFLOW
        Go to Step 13
        [END OF IF]
o   Step 2: SET NEW_NODE = PTR
o   Step 3: SET PTR = PTR -> NEXT
o   Step 4: SET NEW_NODE -> DATA = VAL
o   Step 5: SET TEMP = HEAD
o   Step 6: Repeat Step 7 while TEMP -> NEXT != HEAD
o   Step 7: SET TEMP = TEMP -> NEXT
        [END OF LOOP]
o   Step 8: SET TEMP -> NEXT = NEW_NODE
o   Step 9: SET NEW_NODE -> PREV = TEMP
o   Step 1 : SET NEW_NODE -> NEXT = HEAD
o   Step 11: SET HEAD -> PREV = NEW_NODE
o   Step 12: SET HEAD = NEW_NODE
o   Step 13: 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** by using the following statement.

ptr = (struct node *)malloc(sizeof(struct node));

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. This can be done by using the following statement.

```
head = ptr;
 ptr -> next = head;
 ptr -> prev = head;
```

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.

```
head -> prev = ptr;
 ptr -> next = head;
```

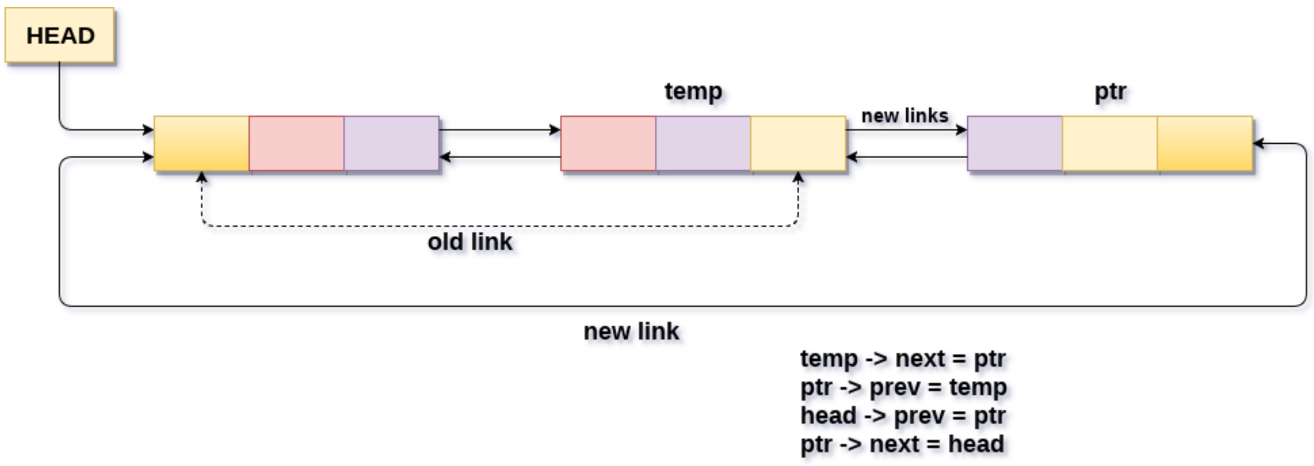
Now, we also need to make the next pointer of the existing last node of the list (temp) point to the new last node of the list, similarly, the new last node will also contain the previous pointer to the temp. this will be done by using the following statements.

```
temp->next = ptr;
 ptr ->prev=temp;
```

In this way, the new node ptr has been inserted as the last node of the list. The algorithm and its C implementation has been described as follows.

Algorithm

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

```
head = NULL;
free(head);
```

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

```
temp = head;
    while(temp -> next != head)
    {
        temp = temp -> next;
    }
```

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. Use the following statement for this purpose.

```
temp -> next = head -> next;
```

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. Use the following statement for this purpose.

```
head -> next -> prev = temp;
```

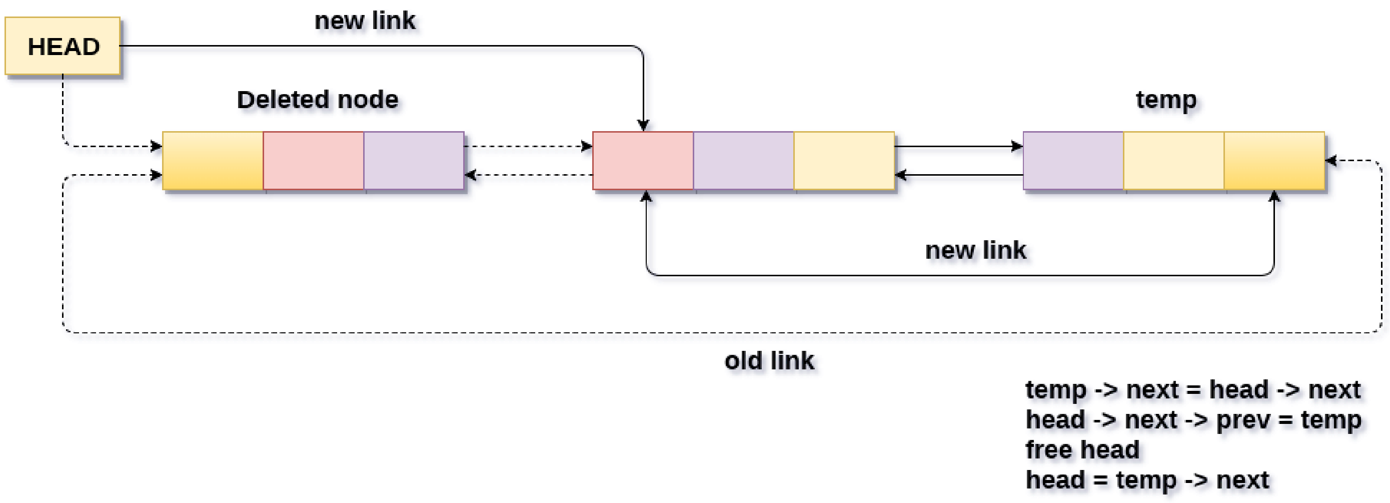
Now, free the head pointer and the make its next pointer, the new head node of the list.

```
free(head);
head = temp -> next;
```

in this way, a node is deleted at the beginning from a circular doubly linked list.

Algorithm

```
o  Step 1: IF HEAD = NULL
       Write UNDERFLOW
       Go to Step 8
       [END OF IF]
o  Step 2: SET TEMP = HEAD
o  Step 3: Repeat Step 4 while TEMP -> NEXT != HEAD
o  Step 4: SET TEMP = TEMP -> NEXT
       [END OF LOOP]
o  Step 5: SET TEMP -> NEXT = HEAD -> NEXT
o  Step 6: SET HEAD -> NEXT -> PREV = TEMP
o  Step 7: FREE HEAD
o  Step 8: SET HEAD = TEMP -> NEXT
```



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

```
head = NULL;
free(head);
```

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

```
temp = head;
 while(temp -> next != head)
 {
    temp = temp -> next;
 }
```

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.

```
temp -> prev -> next = head;
```

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

```
head -> prev = ptr -> prev;
```

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

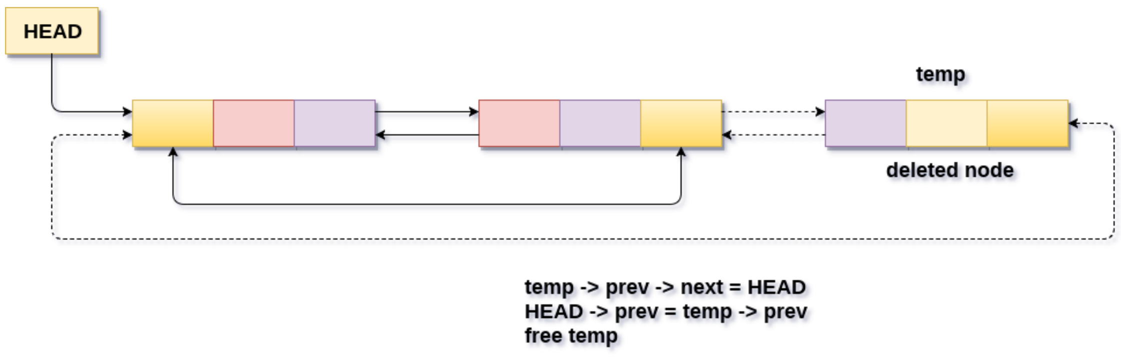
```
free(head)
```

in this way, the last node of the list is deleted.

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



Deletion in circular doubly linked list at beginning

Applications of linked list data structure

Linked lists are widely used in various applications due to their flexibility and efficiency. Here are some common scenarios where linked lists are used:

1. Dynamic Memory Allocation: Linked lists allow for dynamic memory allocation, meaning that memory can be allocated or deallocated as needed during program execution. This makes them suitable for applications where the size of data is unpredictable, such as in text editors, where users can input variable-length text.

2. Implementation of Stacks and Queues: Linked lists serve as the underlying data structure for implementing stacks and queues. In a stack, elements are added and removed from one end (top), while in a queue, elements are added at one end (rear) and removed from the other end (front). Linked lists offer efficient insertion and deletion operations, making them ideal for implementing these data structures.

3. Polynomial Representation: Linked lists are commonly used to represent polynomials in mathematical computations. Each node in the linked list can store a term of the polynomial, containing coefficients and exponents. This representation allows for easy manipulation of polynomial expressions, such as addition, subtraction, and multiplication.

4. File Systems: Linked lists are employed in file systems to maintain the structure of directories and files. Each node in the linked list represents a file or directory, containing information such as file name, size, and pointers to the next file or directory. This hierarchical structure allows for efficient navigation and management of files and directories within the system.

5. Undo Functionality: Linked lists are used to implement undo functionality in software applications. Each action performed by the user is stored as a node in the linked list, allowing users to undo their actions by traversing the list in reverse order. This provides users with the ability to revert back to previous states of the application, enhancing user experience and productivity.

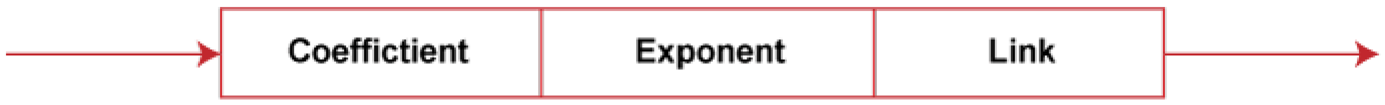
Polynomial Manipulation

Polynomial manipulations are one of the most important applications of linked lists. Polynomials are an important part of mathematics not inherently supported as a data type by most languages. A polynomial is a collection of different terms, each comprising coefficients, and exponents. It can be represented using a linked list. This representation makes polynomial manipulation efficient.
While representing a polynomial using a linked list, each polynomial term represents a node in the linked list. To get better efficiency in processing, we assume that the term of every polynomial is stored within the linked list in the order of decreasing exponents. Also, no two terms have the same exponent, and no term has a zero coefficient and without coefficients. The coefficient takes a value of 1.

Each node of a linked list representing polynomial constitute three parts:

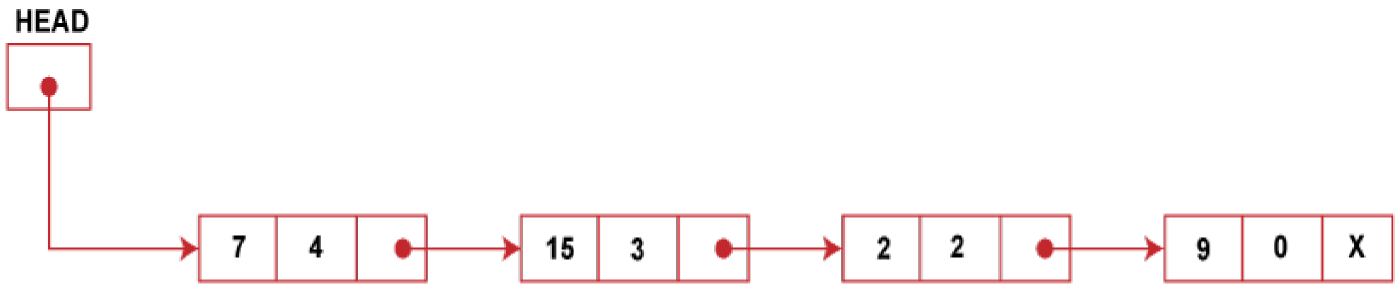
- The first part contains the value of the coefficient of the term.
- The second part contains the value of the exponent.
- The third part, LINK points to the next term (next node).

The structure of a node of a linked list that represents a polynomial is shown below:



Node representing a term of a polynomial

Consider a polynomial P(x) = $7x^2 + 15x^3 - 2\ x^2 + 9$. Here 7, 15, -2, and 9 are the coefficients, and 4,3,2,0 are the exponents of the terms in the polynomial. On representing this polynomial using a linked list, we have



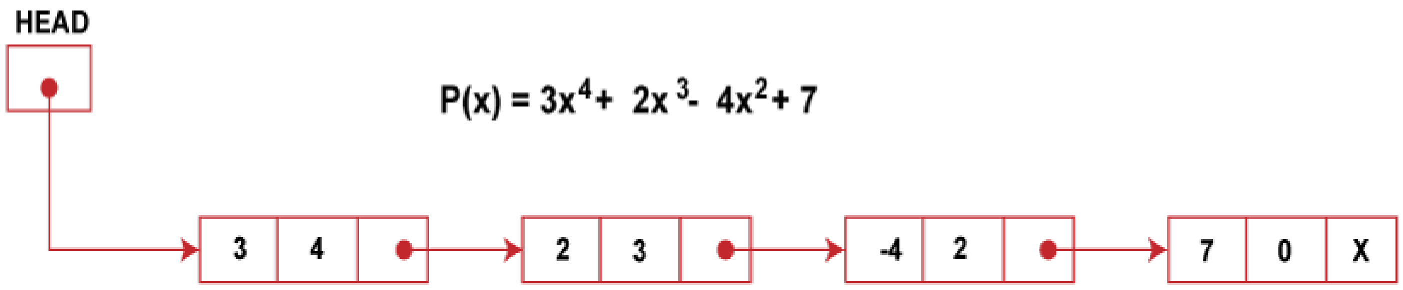
Linked representation of polynomial P(x)

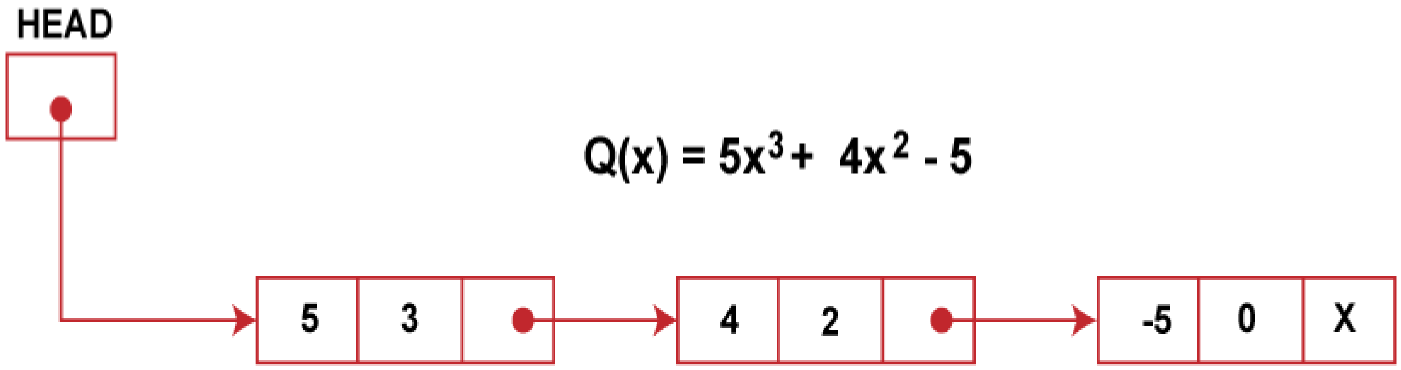
Observe that the number of nodes equals the number of terms in the polynomial. So we have 4 nodes. Moreover, the terms are stored to decrease exponents in the linked list. Such representation of polynomial using linked lists makes the operations like subtraction, addition, multiplication, etc., on polynomial very easy.

Addition of Polynomials:

To add two polynomials, we traverse the list P and Q. We take corresponding terms of the list P and Q and compare their exponents. If the two exponents are equal, the coefficients are added to create a new coefficient. If the new coefficient is equal to 0, then the term is dropped, and if it is not zero, it is inserted at the end of the new linked list containing the resulting polynomial. If one of the exponents is larger than the other, the corresponding term is immediately placed into the new linked list, and the term with the smaller exponent is held to be compared with the next term from the other list. If one list ends before the other, the rest of the terms of the longer list is inserted at the end of the new linked list containing the resulting polynomial.
Let us consider an example an example to show how the addition of two polynomials is performed,

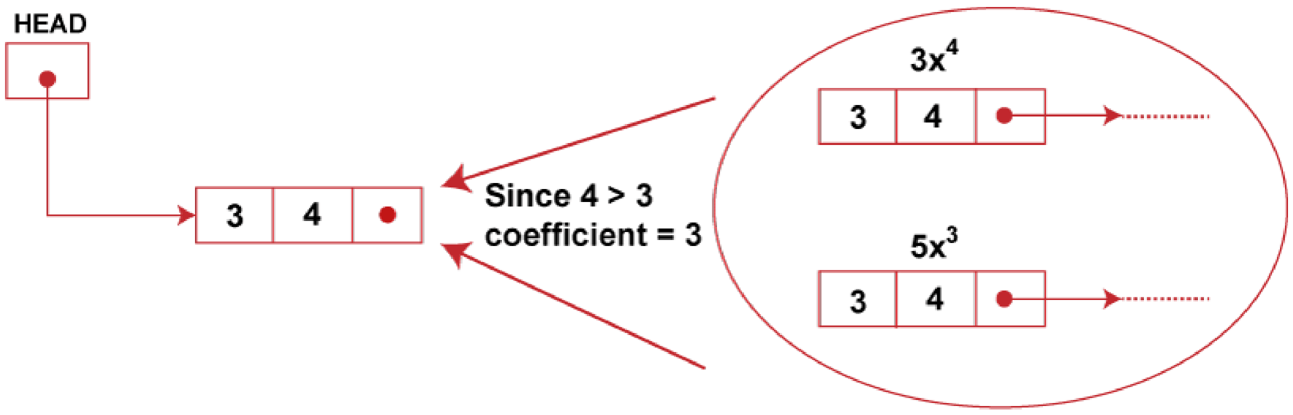
$P(x) = 3x^4 + 2x^3 - 4x^2 + 7$
$Q(x) = 5x^3 + 4x^2 - 5$
These polynomials are represented using a linked list in order of decreasing exponents as follows:





To generate a new linked list for the resulting polynomials that is formed on the addition of given polynomials P(x) and Q(x), we perform the following steps,

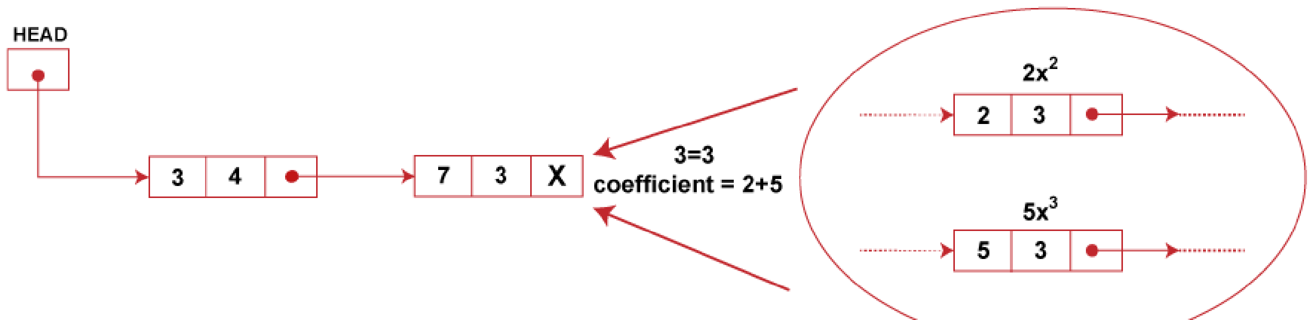
1. Traverse the two lists P and Q and examine all the nodes.
2. We compare the exponents of the corresponding terms of two polynomials. The first term of polynomials P and Q contain exponents 4 and 3, respectively. Since the exponent of the first term of the polynomial P is greater than the other polynomial Q, the term having a larger exponent is inserted into the new list. The new list initially looks as shown below:



Term with coefficient 3, exponent 4 added

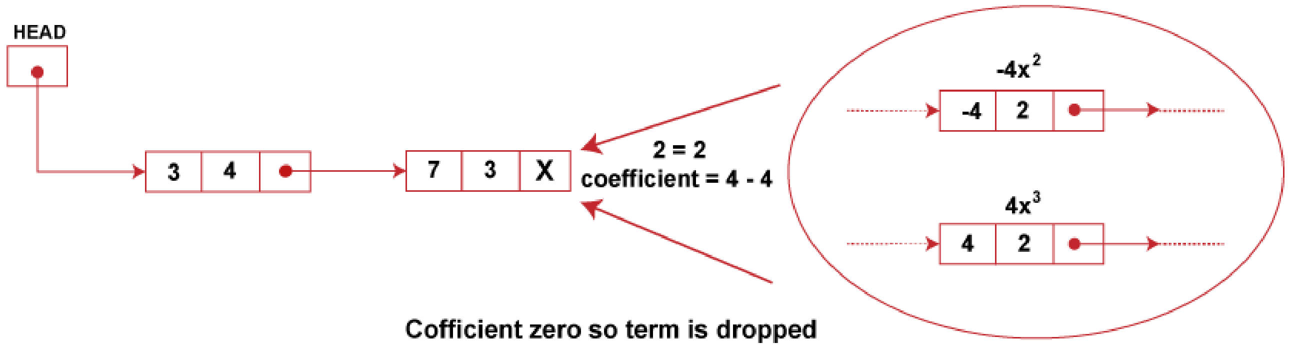
3. We then compare the exponent of the next term of the list P with the exponents of the present term of list Q. Since the two exponents are equal, so their coefficients are added and

appended to the new list as follows:



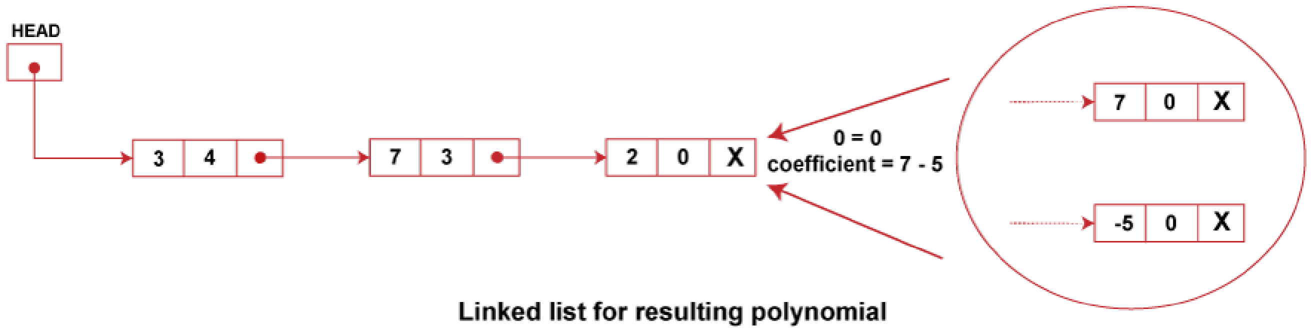
Term with coefficient 7, exponent 3 added

4. Then we move to the next term of P and Q lists and compare their exponents. Since exponents of both these terms are equal and after addition of their coefficients, we get 0, so the term is dropped, and no node is appended to the new list after this,



Cofficient zero so term is dropped

5. Moving to the next term of the two lists, P and Q, we find that the corresponding terms have the same exponents equal to 0. We add their coefficients and append them to the new list for the resulting polynomial as shown below:



Linked list for resulting polynomial