2. LINKED LISTS

UNIT II

Linked Lists: Singly linked lists: representation and operations, doubly linked lists and circular linked lists, Comparing arrays and linked lists, Applications of linked lists.

Introduction

A linked list is a collection of data elements called nodes in which the linear representation is given by links from one node to the next node. A linked list does not store its elements in consecutive memory locations and the user can add any number of elements to it.

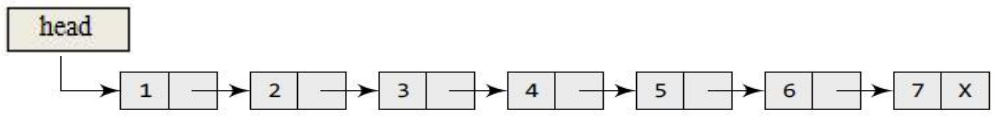
The elements in a linked list can be accessed only in a sequential manner. But like an array, insertions and deletions can be done at any point in the list in a constant time.

A linked list, in simple terms, is a linear collection of data elements. These data elements are called nodes. Linked list is a data structure which in turn can be used to implement other data structures.

Thus, it acts as a building block to implement data structures such as stacks, queues, and their variations.

Basic Terminology:

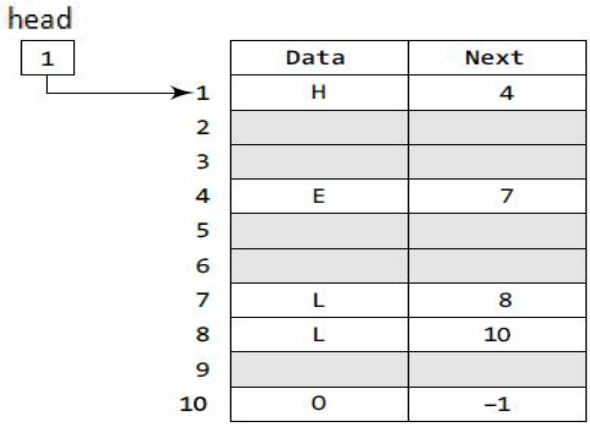
A linked list can be perceived as a train or a sequence of nodes in which each node contains one or more data fields and a pointer to the next node.



we can see a linked list in which every node contains two parts, an integer and a pointer to the next node. The last node will have no next node connected to it, so it will store a special value called NULL.

Since in a linked list, every node contains a pointer to another node which is of the same type, it is also called a self-referential data type.

Let us see how a linked list is maintained in the memory. When we traverse DATA and NEXT in this manner, we finally see that the linked list in the above example stores characters that when put together form the word HELLO.



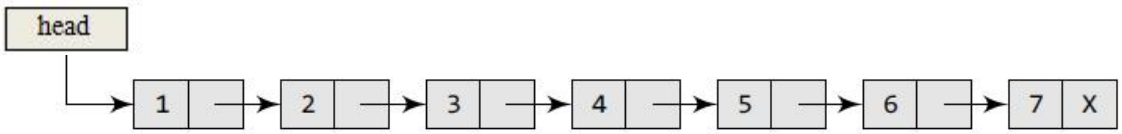
Linked Lists versus Arrays:

Both arrays and linked lists are a linear collection of data elements. But unlike an array, a linked list does not store its nodes in consecutive memory locations. Another point of difference between an array and a linked list is that a linked list does not allow random access of data. Nodes in a linked list can be accessed only in a sequential manner.

Another advantage of a linked list over an array is that we can add any number of elements in the list. This is not possible in case of an array.

Single Linked Lists:

A singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.

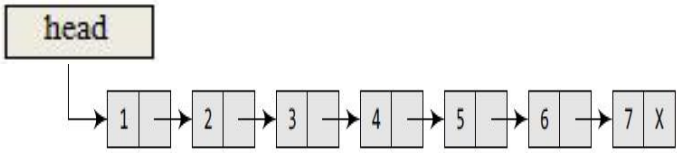


Traversing a linked list means accessing the nodes of the list in order to perform some processing on them. Remember a linked list always contains a pointer variable START which stores the address of the first node of the list. End of the list is marked by storing NULL or –1 in the NEXT field of the last node.

Operations

Traversing a Linked List:

For traversing the linked list, we also make use of another pointer variable PTR which points to the node that is currently being accessed. Algorithm for traversing a linked list

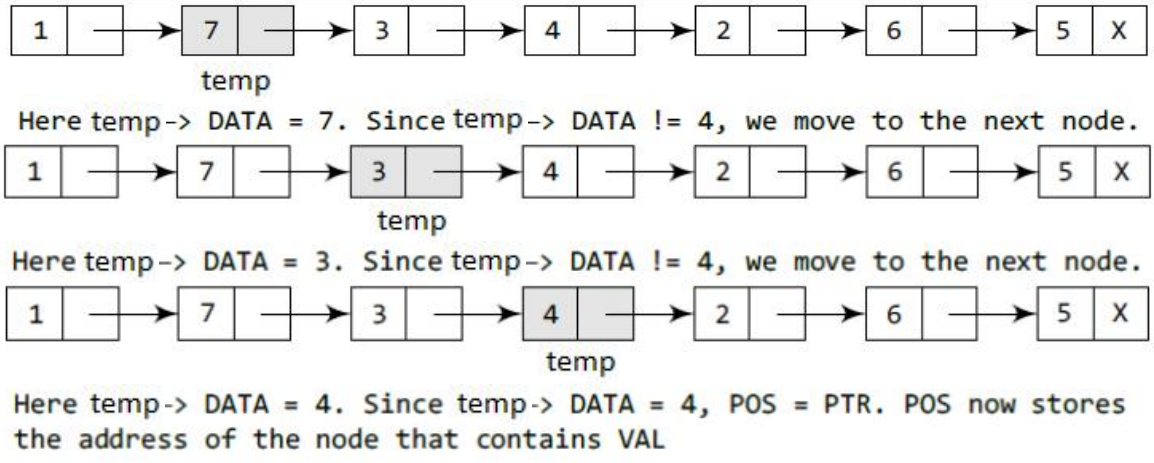


Step-1: check list is empty are not (head == NULL)
Step-2: if empty print list is empty and goto step 8
Step-3: otherwise define a pointer temp and allocate to head
Step-4: Repeat steps 5,6 while temp->next !=NULL
Step-5: print temp->data
Step-6: temp=temp->next
End of Loop
Step-7: print temp->data
Step-8: exit

Searching for a Value in a Linked List:

Searching a linked list means to find a particular element in the linked list. So searching means finding whether a given value is present in the information part of the node or not. If it is present, the algorithm returns the address of the node that contains the value. However, if the search is unsuccessful, POS is set to NULL which indicates that VAL is not present in the linked list.

Consider the linked list shown in below. If we have VAL = 4, then the flow of the algorithm can be explained as shown in the figure.



Step-1: check list is empty are not (head == NULL)
Step-2: if empty print list is empty and goto step 8
Step-3: otherwise define a pointer temp and allocate to head
Step-4: Repeat steps 5-7 while temp !=NULL
Step-5: check if temp->data ==value
Step-6: then print "Key found" and goto step 8
Step-7: otherwise set temp=temp->next
End of Loop
Step-8: Exit

Inserting a New Node in a Linked List:

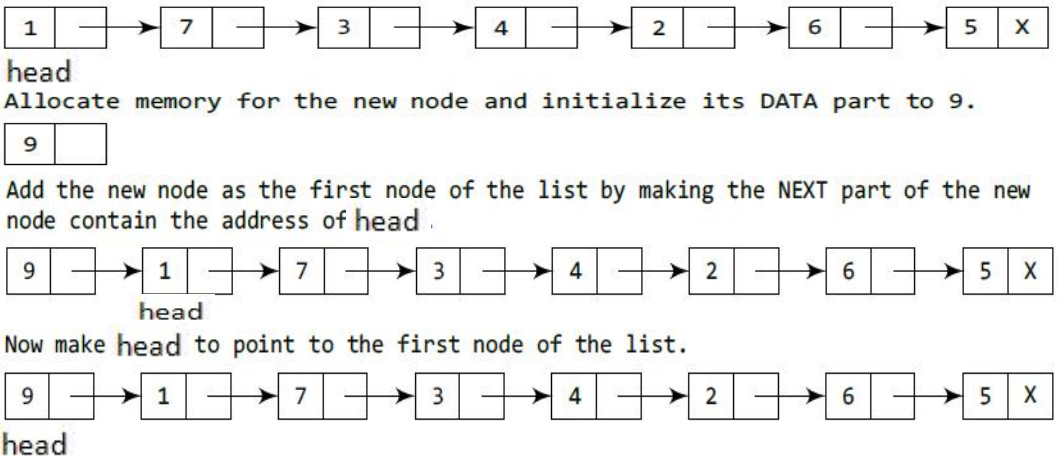
we will see how a new node is added into an already existing linked list. We will take four cases and then see how insertion is done in each case.

Case 1: The new node is inserted at the beginning.
Case 2: The new node is inserted at the end.
Case 3: The new node is inserted after a given node.

Let us first discuss an important term called UNDERFLOW. Underflow is a condition that occurs when head = NULL or no memory cell is present in the system. When this condition occurs, the program must give an appropriate message.

Case 1: Inserting a Node at the Beginning of a Linked List

Inserting a Node at the Beginning of a Linked List. Consider the linked list shown in below figure. Suppose we want to add a new node with data 9 and add it as the first node of the list.



step-1:Define the newnode (struct node *newnode)
step-2:Allocate memory for the new node
step-3set newnode->data=value
step-4: Check if the list is empty or not(head==NULL)
step-5:if empty set newnode->next= NULL
step-6: set head=newnode and go to step 9
step-7: If it is not empty set newnode->next=head
step-8: set head=newnode
Step-9: print one node inserted.

Case 2: Inserting a Node at the End of a Linked List

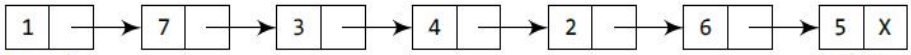
head

Allocate memory for the new node and initialize its DATA part to 9 and NEXT part to NULL.

Take a pointer variable temp which points to head

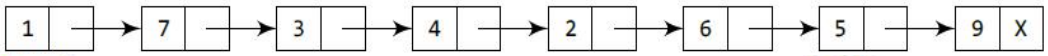
head , temp

Move temp so that it points to the last node of the list.



head temp

Add the new node after the node pointed by temp This is done by storing the address of the new node in the NEXT part of temp

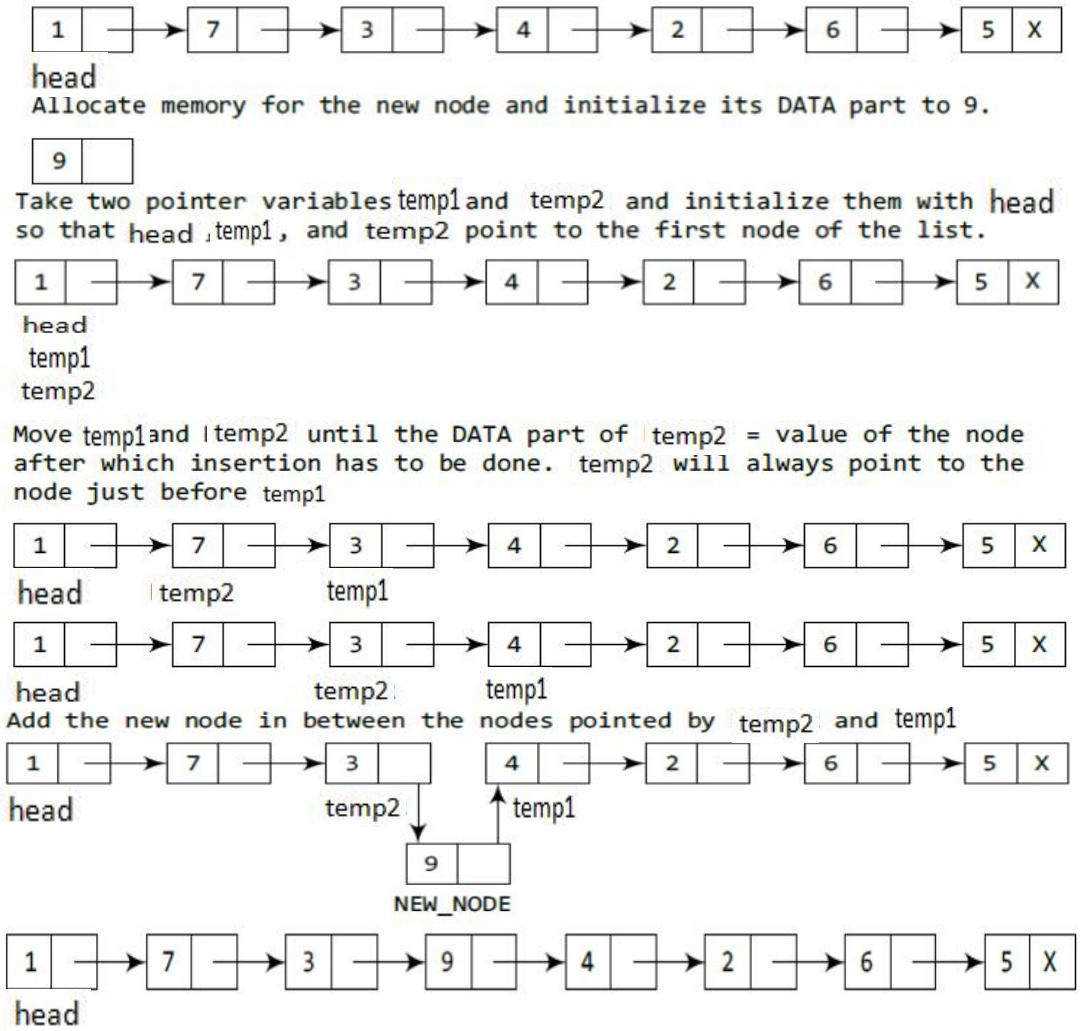


head temp

step-1: Define the newnode (struct node *newnode)
step-2: Allocate memory for the new node
step-3: set newnode->data=value
step-4: set newnode->next= NULL
step-5: Check if the list is empty or not(head==NULL)
step-6: if empty set head->newnode and go to step 11
step-7: If it is not empty define a pointer temp and allocate to head
step-8: Repeat step 9 while temp->next !=NULL
Step-9: temp=temp->next
Step-10: set temp->next=newnode
Step-11: print one node inserted.

Case 3: Inserting a Node After a Given Node in a Linked List

Consider the linked list shown in below figure. Suppose we want to add a new node with value 9 after the node containing 3.



step-1: Define the newnode (struct node *newnode)
step-2: Allocate memory for the new node
step-3: set newnode->data=value
step-4: Check if the list is empty or not(head==NULL)
step-5: if empty set newnode->next= NULL
step-6: set head->newnode and go to step 12
step-7: If it is not empty define a pointer temp and allocate to head
step-8: Repeat step 9 while temp->data!=loc1 && temp->data!=loc2
Step-9: temp=temp->next
Step-10: set newnode->next=temp
Step-11: set temp->next=newnode
Step- 12:print one node inserted.

Deleting a Node from a Linked List:

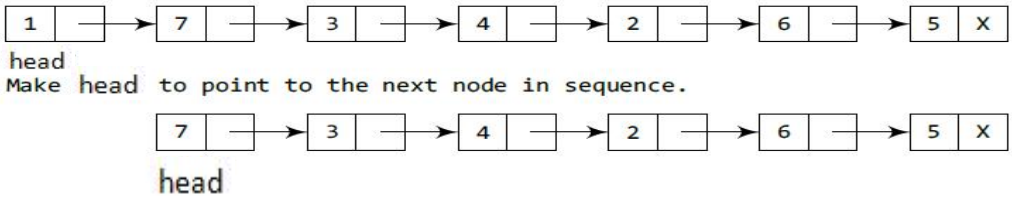
We will discuss how a node is deleted from an already existing linked list. We will consider three cases and then see how deletion is done in each case.

Case 1: The first node is deleted.

Case 2: The last node is deleted.

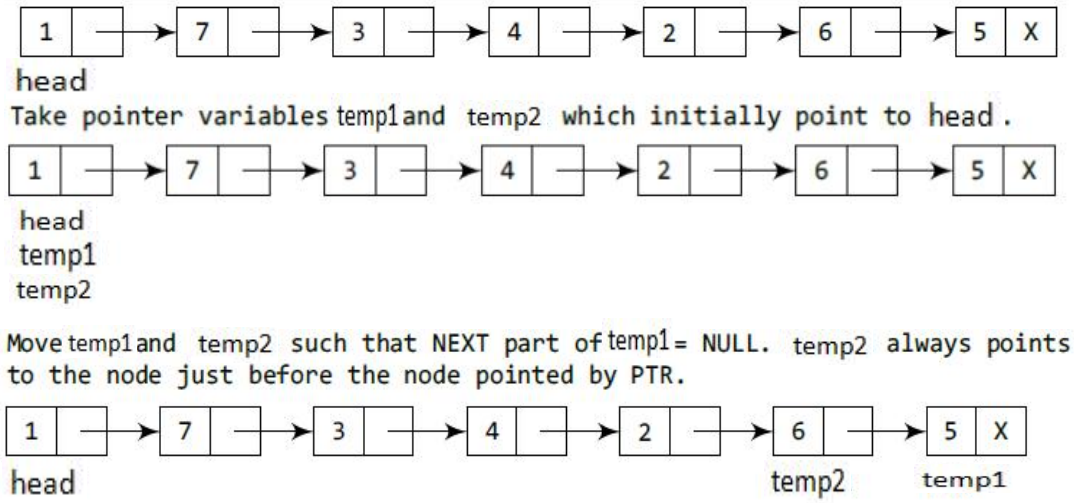
Case 3: The node after a given node is deleted.

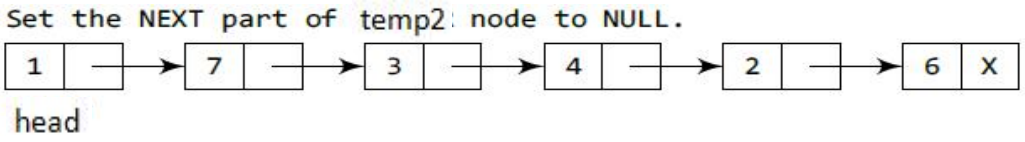
Case 1: Deleting a First Node from a Linked List



Step-1: check list is empty are not (head == NULL)
Step-2: if empty print "list is empty" and goto 10.
Step-3: if it is not empty check head->next==NULL
Step-4: free(head)
Step-5: set head=NULL and goto step 9.
Step-6: otherwise define a pointer temp and allocate to head
Step-7: set head=head->next
Step-8: free(temp)
Step-9: print one node deleted.
Step-10:exit

Case 2: Deleting the Last Node from a Linked List

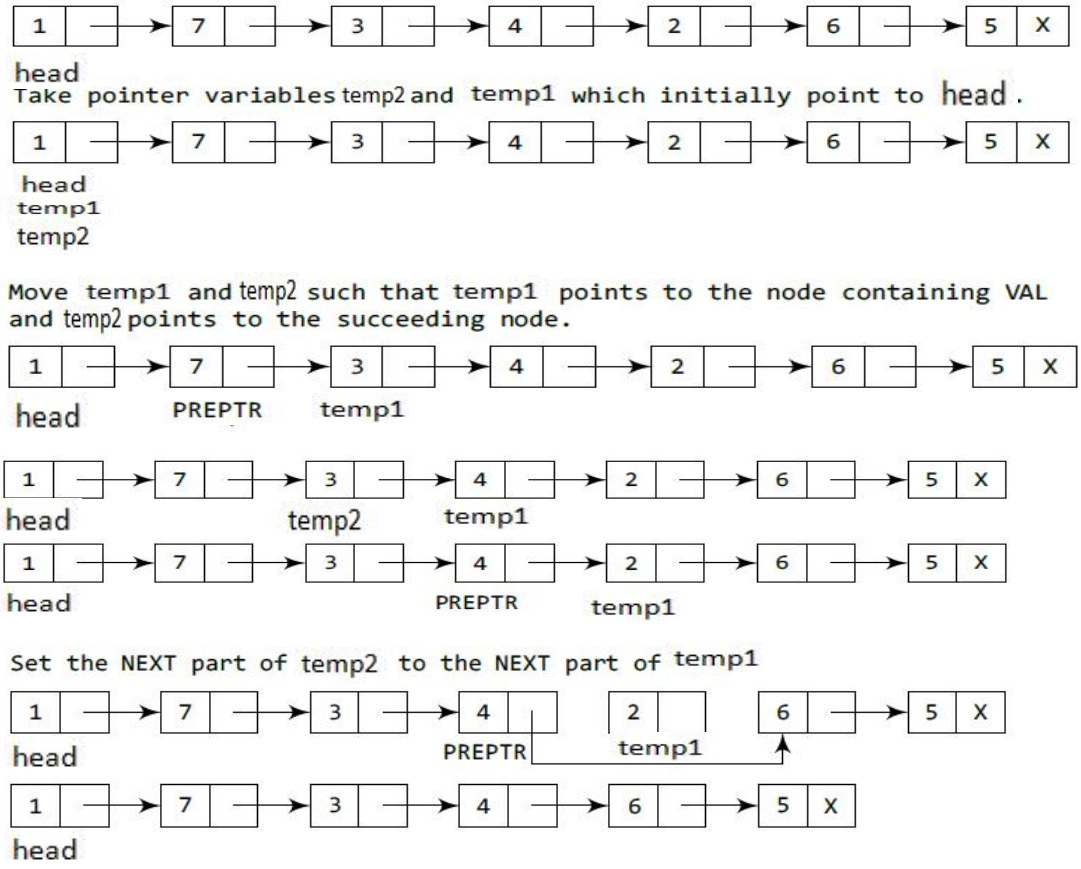




Step-1: check list is empty are not (head == NULL)
Step-2: if empty print list is empty and goto step 13
Step-3: if it is not empty, then check head->next==NULL
Step-4: free(head)
Step-5: set head=NULL and goto step 12.
Step-6: otherwise define two pointers temp1, temp2 and allocate temp1 to head
Step-7: Repeat step 8,9 while temp1->next !=NULL
Step-8: temp2=temp1
Step-9: temp1=temp1->next
End Loop
Step-10: set temp2->next=NULL
Step-11: free(temp1)
Step-12: print one node deleted.
Step-13: exit.

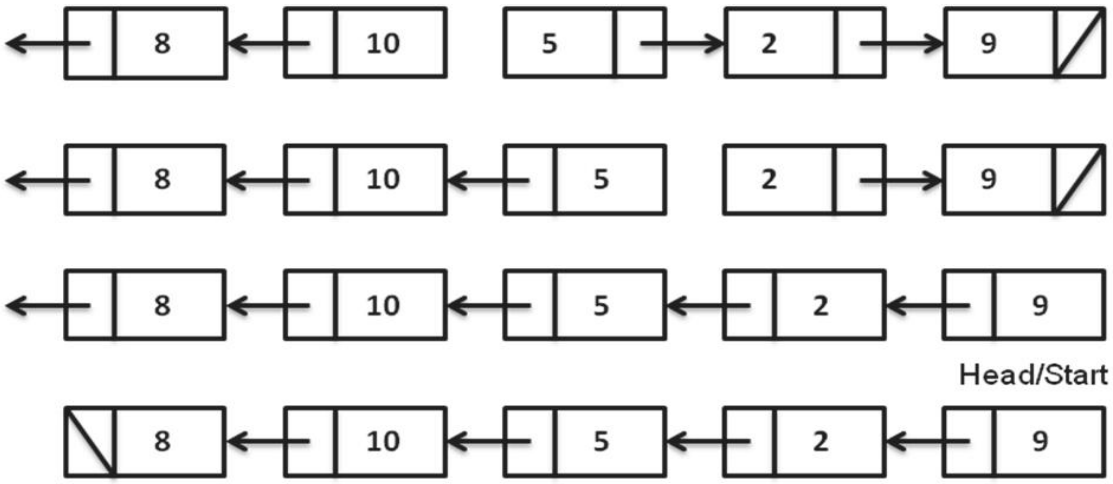
Case 3: Deleting After a Given Node in a Linked List

Consider the linked list shown in below figure. Suppose we want to delete the node that succeeds the node which contains data value 4.



Step-1: check list is empty are not (head == NULL)
Step-2: if empty print list is empty and goto step 12
Step-3: otherwise define two pointers temp1, temp2 and allocate temp1 to head
Step-4: Repeat steps 5-8 while temp1->data !=value
Step-5: check if temp1->next==NULL
Step-6: then print "key does not exist" and goto step 12
Step-7: temp2=temp1
Step-8: temp1=temp1->next
End of Loop
Step-9: set temp2->next=temp1->next
Step-10: free(temp1)
Step-11: print one node deleted.
Step-12: exit.

Reversing Single Linked list:




```
 Void reverse()
 {
 struct node *prevNode=null, *temp=head;
while(temp != NULL)

{    head = head->next;
temp->next = prevNode;

 prevNode = temp;

temp = head;

}      head = prevNode;

}
```

Applications on Single Linked list

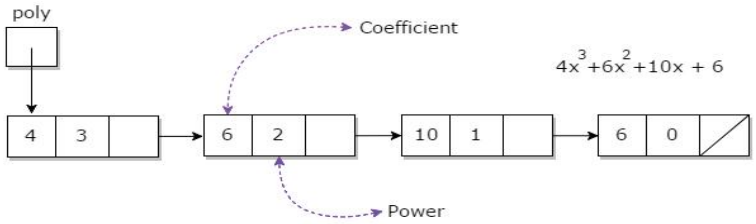
- Implementation of stacks and queues.
- Implementation of graphs: Adjacency list representation of graphs is most popular which uses linked list to store adjacent vertices.
- Dynamic memory allocation: We use linked list of free blocks.
- Maintaining directory of names. Performing arithmetic operations on long integers
- Manipulation of polynomials by storing constants in the node of linked list. Representing sparse matrices

Polynomial Expression Representation:

A polynomial is composed of different terms where each of them holds a coefficient and an exponent. An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts: one is the coefficient and other is the exponent

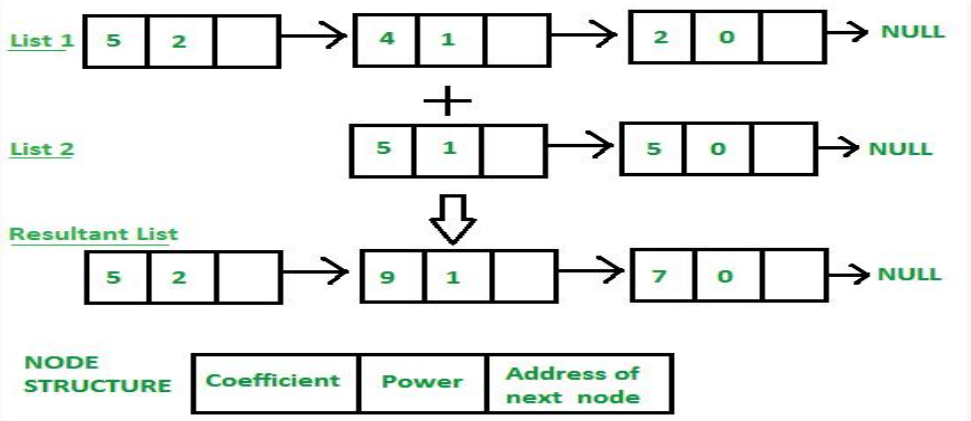
Example:

$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 are its exponential values.



Addition of two polynomials:

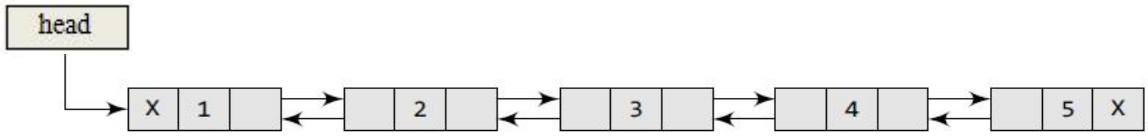
```
Input:
      1st number = 5x^2 + 4x^1 + 2x^0
      2nd number = 5x^1 + 5x^0
Output:
          5x^2 + 9x^1 + 7x^0
Input:
      1st number = 5x^3 + 4x^2 + 2x^0
      2nd number = 5x^1 + 5x^0
Output:
          5x^3 + 4x^2 + 5x^1 + 7x^0
```



Doubly linked list

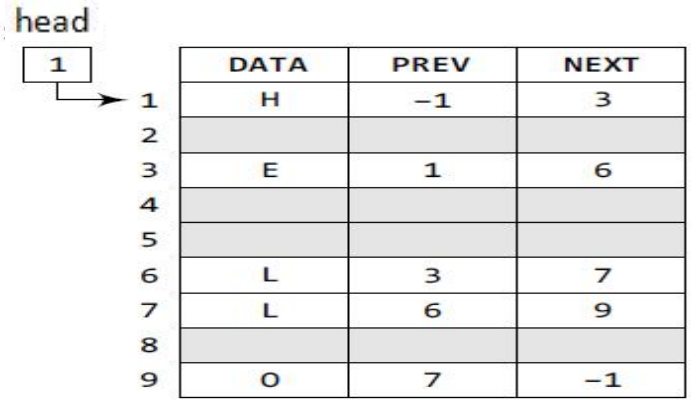
A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence.

Therefore, it consists of three parts—data, a pointer to the next node, and a pointer to the previous node.



A doubly linked list provides the ease to manipulate the elements of the list as it maintains pointers to nodes in both the directions (forward and backward).

The main advantage of using a doubly linked list is that it makes searching twice as efficient. Let us view how a doubly linked list is maintained in the memory.



	DATA	PREV	NEXT
1	H	−1	3
2			
3	E	1	6
4			
5			
6	L	3	7
7	L	6	9
8			
9	O	7	−1

Inserting a New Node in a Doubly Linked List:

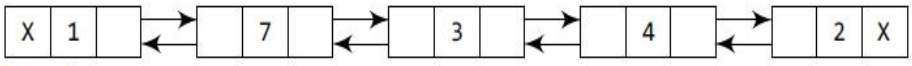
In this section, we will discuss how a new node is added into an already existing doubly linked list. We will take four cases and then see how insertion is done in each case.

Case 1: The new node is inserted at the beginning.

Case 2: The new node is inserted at the end.

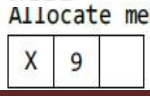
Case 3: The new node is inserted after a given node.

Case 1: Inserting a Node at the Beginning of a Doubly Linked List

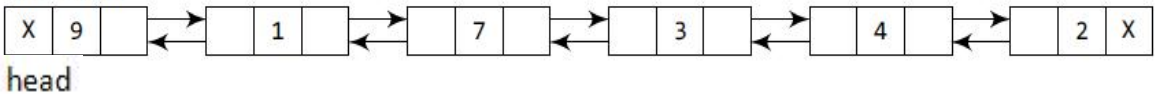


head

Allocate memory for the new node and initialize its DATA part to 9 and PREV field to NULL.



Add the new node before the head node. Now the new node becomes the first node of the list.



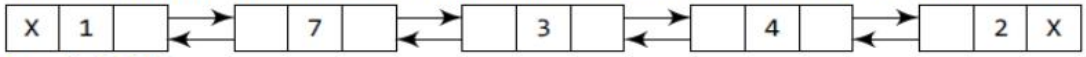
Step-1: check list is empty are not (head == NULL)
Step-2: if empty print list is empty and goto step 8
Step-3: otherwise define a pointer temp and allocate to head
Step-4: Repeat steps 5,6 while temp->next !=NULL
Step-5: print temp->data
Step-6: temp1=temp1->next
End of Loop
Step-7: print temp->data
Step-8: exit

Case 2: Inserting a Node at the end of a Doubly Linked List

head

Allocate memory for the new node and initialize its DATA part to 9 and its NEXT field to NULL.

Take a pointer variable temp and make it point to the first node of the list.



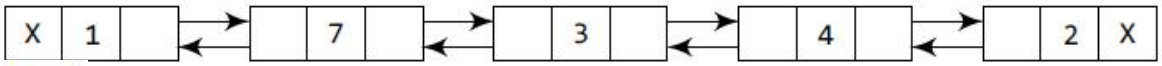
head , temp

Move temp so that it points to the last node of the list. Add the new node after the node pointed by temp

head temp

step-1: Define the newnode (struct node *newnode)
step-2: Allocate memory for the new node
step-3: set newnode->data=value
step-4: set newnode->next=NULL
step-5: Check if the list is empty or not(head==NULL)
Step-6: if empty set newnode->prev=NULL
step-7: set head=newnode and go to step 13
step-8: otherwise define a pointer temp and allocate to head
step-9: Repeat step 10 while temp->next!=NULL
Step-10: temp=temp->next
Step-11: set newnode->prev=temp
Step-12: set temp->next=newnode
Step- 13:print one node inserted.

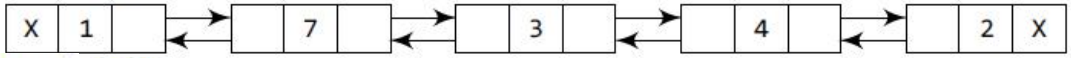
Case 3: Inserting a Node After a Given Node in a Doubly Linked List



head

Allocate memory for the new node and initialize its DATA part to 9.

Take a pointer variable temp and make it point to the first node of the list.

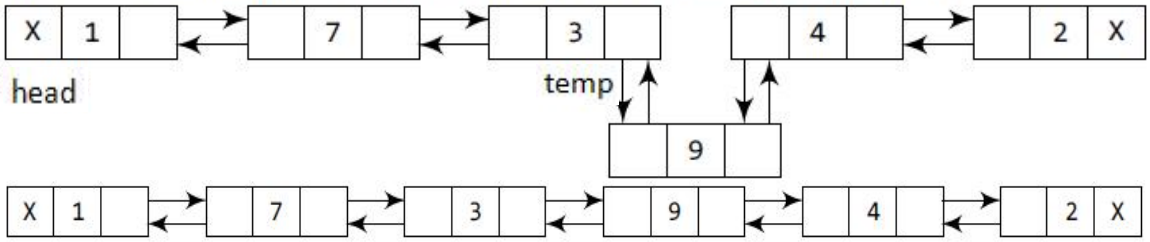


head , temp

Move temp further until the data part of temp= value after which the node has to be inserted.

head temp

Insert the new node between temp and the node succeeding it.



head

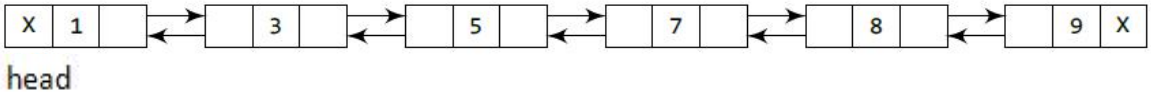
step-1: Define the newnode (struct node *newnode)
step-2: Allocate memory for the new node
step-3: set newnode->data=value
step-4: Check if the list is empty or not(head==NULL)
step-5: if empty set newnode->prev=NULL
Step-6: set newnode->next=NULL
step-7: set head=newnode and go to step 15
step-8: otherwise define a pointer temp and allocate to head
step-9: Repeat step 10 while temp->data!=loc1 && temp->data!=loc2
Step-10: temp=temp->next
Step-11: set newnode->prev=temp
Step-12: set newnode->next=temp->next
Step- 13:set newnode->next->prev=newnode
Step-14: set temp->next=newnode
Setp-15: print one node inserted.

Deleting a Node from a Doubly Linked List

In this section, we will see how a node is deleted from an already existing doubly linked list. We will take four cases and then see how deletion is done in each case.

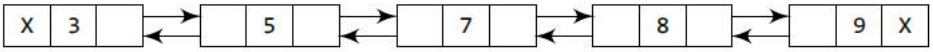
Case 1: The first node is deleted.
Case 2: The last node is deleted.
Case 3: The node after a given node is deleted.

Case 1: Deleting the First Node from a Doubly Linked List



head

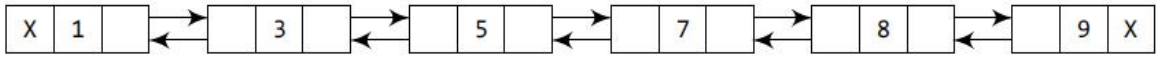
Free the memory occupied by the first node of the list and make the second node of the list as the head node.



head

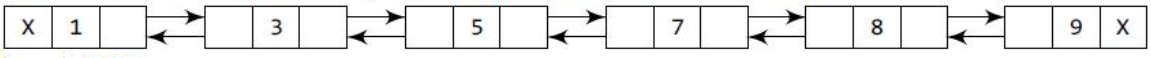
Step-1: check list is empty are not (head == NULL)
Step-2: if empty print "list is empty" and goto 11.
Step-3: if it is not empty check if head->next==head
Step-4: free(head)
Step-5: set head=NULL and goto step 10.
Step-6: otherwise define two pointers temp and allocate to head
Step-7: set head=head->next
Step-8: set head->prev=NULL
Step-9: free(temp)
Step-10: print one node deleted
step-11: exit.

Case 2: Deleting the Last Node from a Doubly Linked List



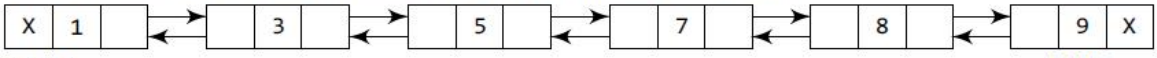
head

Take a pointer variable temp that points to the first node of the list.



head , temp

Move temp so that it now points to the last node of the list.



temp

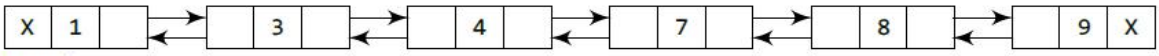
head

Free the space occupied by the node pointed by temp and store NULL in NEXT field of its preceding node.

head

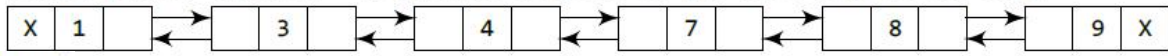
Step-1: check list is empty are not (head == NULL)
Step-2: if empty print list is empty and goto step 12
Step-3: if it is not empty, then check head->next==NULL
Step-4: free(head)
Step-5: set head=NULL and goto step 11.
Step-6: otherwise define a pointers temp and allocate to head
Step-7: Repeat step 8 while temp->next !=NULL
Step-8: temp=temp->next
Step-9: temp->prev->next=NULL
Step-10: free(temp)
Step-11: print one node deleted.
Step-12: exit.

Case 3: Deleting the Node After a Given Node in a Doubly Linked List



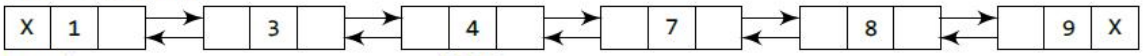
head

Take a pointer variable temp and make it point to the first node of the list.



head , temp

Move temp further so that its data part is equal to the value after which the node has to be inserted.



head temp

Delete the node succeeding temp

head temp

head

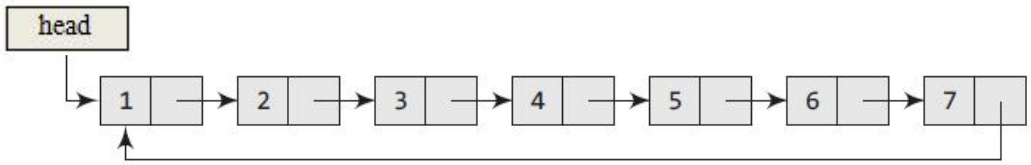
Circular Linked List

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

While traversing a circular linked list, we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node where we started. Thus, a circular linked list has no beginning and no ending.

Note that there are no NULL values in the NEXT part of any of the nodes of list.

Step-1: check list is empty are not (head == NULL)
Step-2: if empty print list is empty and goto step 8
Step-3: otherwise define a pointer temp and allocate to head
Step-4: Repeat steps 5,6 while temp->next !=head
Step-5: print temp->data
Step-6: temp1=temp1->next
End of Loop
Step-7: print temp->data
Step-8: exit



Operation:

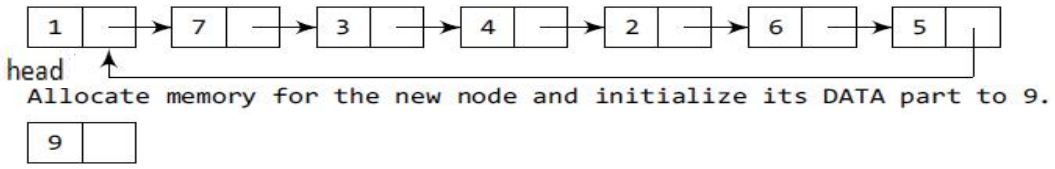
Inserting a New Node in a Circular Linked List

In this section, we will see how a new node is added into an already existing linked list. We will take two cases and then see how insertion is done in each case.

Case 1: The new node is inserted at the beginning of the circular linked list.

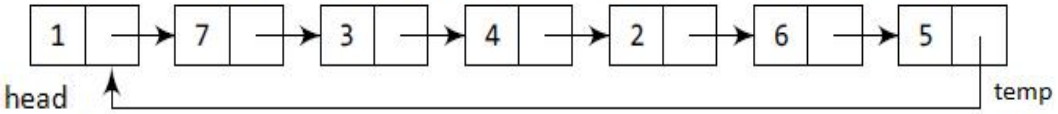
Case 2: The new node is inserted at the end of the circular linked list.

Case 1: Inserting a Node at the Beginning of a Circular Linked List

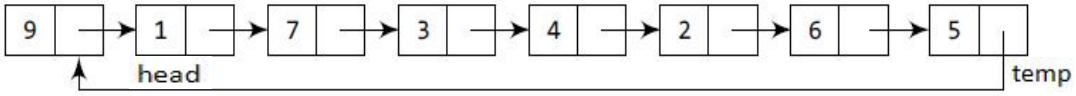


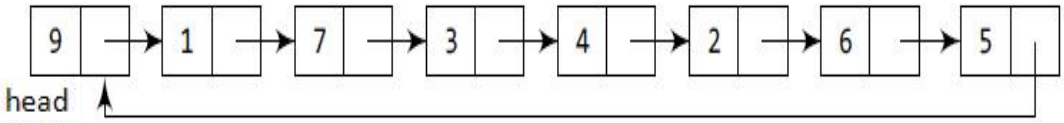
Take a pointer variable temp that points to the head node of the list.

Move temp so that it now points to the last node of the list.



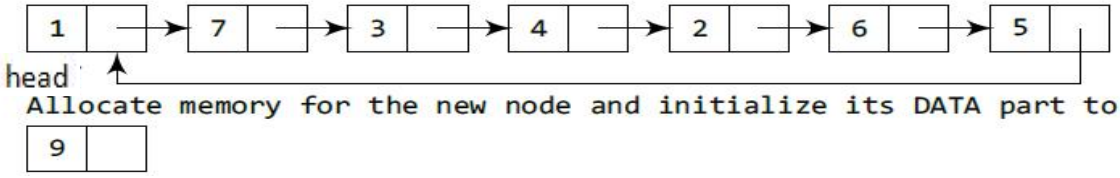
Add the new node in between temp and head .

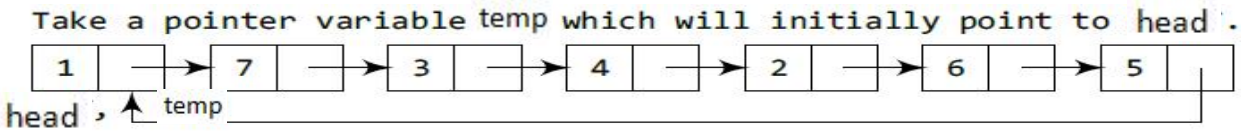


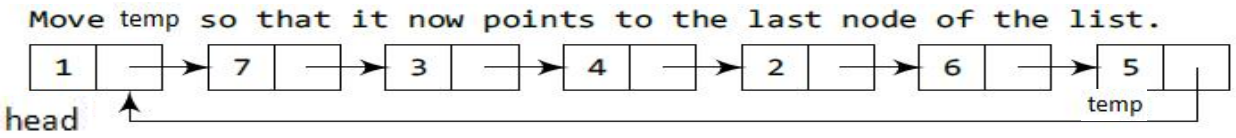


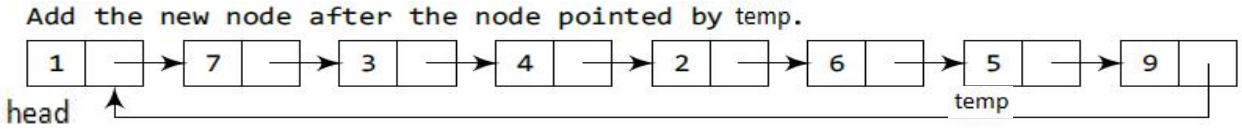
step-1:Define the newnode (struct node *newnode)
step-2:Allocate memory for the new node
step-3 set newnode->data=value
step-4: Check if the list is empty or not(head==NULL)
step-5:if empty set newnode->next= newnode
step-6: set head=newnode and go to step 13
step-7: If it is not empty define a pointer temp and allocate to head
step-8: Repeat step 9 while temp->next !=head
Step-9: temp=temp->next
Step-10: set newnode->next=head
Step-11: set temp->next=newnode
Step-12: set newnode=head
Step-13: print one node inserted.

Case 2: Inserting a Node at the End of a Circular Linked List









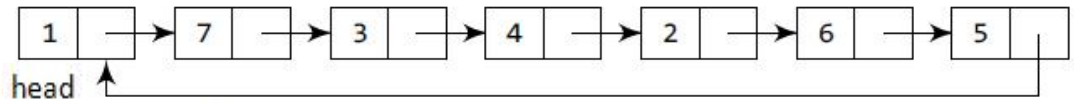
step-1:Define the newnode (struct node *newnode)
step-2:Allocate memory for the new node
step-3 set newnode->data=value
step-4: Check if the list is empty or not(head==NULL)
step-5:if empty set newnode->next= newnode
step-6: set head=newnode and go to step 12
step-7: If it is not empty define a pointer temp and allocate to head
step-8: Repeat step 9 while temp->next !=head
Step-9: temp=temp->next
Step-10: set newnode->next=head
Step-11: set temp->next=newnode
Step-12: print one node inserted.

Deleting a Node from a Circular Linked List

In this section, we will discuss how a node is deleted from an already existing circular linked list. We will take two cases and then see how deletion is done in each case. Rest of the cases of deletion are same as that given for singly linked lists.

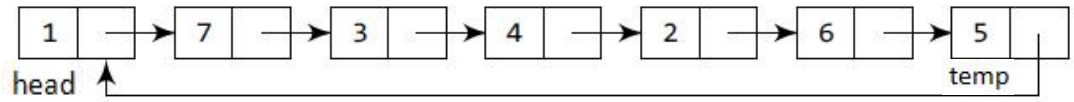
Case 1: The first node is deleted.
Case 2: The last node is deleted.

Case 1: Deleting the First Node from a Circular Linked List

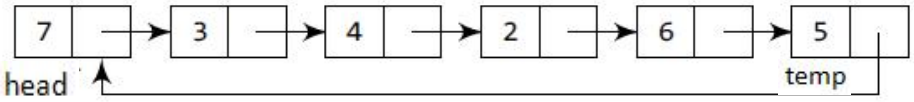


Take a variable temp and make it point to the head node of the list.

Move temp further so that it now points to the last node of the list.

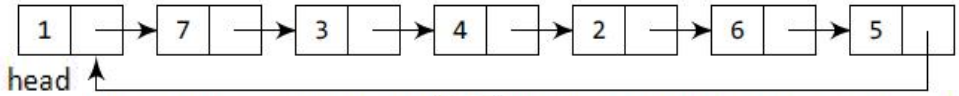


The NEXT part of temp is made to point to the second node of the list and the memory of the first node is freed. The second node becomes the first node of the list.

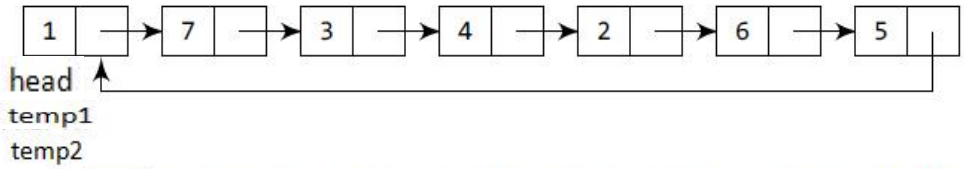


Step-1: check list is empty are not (head == NULL)
Step-2: if empty print "list is empty" and goto 10.
Step-3: if it is not empty check if head->next==head
Step-4: free(head)
Step-5: set head=NULL and goto step 12.
Step-6: otherwise define two pointers temp1, temp2 and allocate to head
Step-7: Repeat step 8 while temp1->next !=head
Step-8: temp1=temp1->next
Step-9: set head=head->next
Step-10: set temp1->next=head
step-11: free(temp2)
Step-12: print one node deleted.
Step-13: exit.

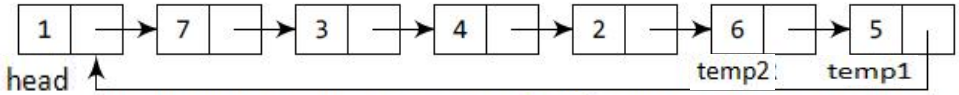
Case 2: Deleting the Last Node from a Circular Linked List



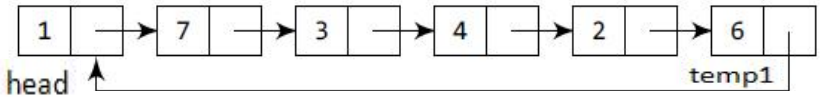
Take two pointers temp2 and temp1 which will initially point to head .



Move temp1 so that it points to the last node of the list. temp2 will always point to the node preceding temp1



Make the temp2 's next part store head node's address and free the space allocated for temp1 Now temp2 is the last node of the list.



Step-1: check list is empty are not (head == NULL)
Step-2: if empty print list is empty and goto step 13
Step-3: if it is not empty, then check head->next==NULL
Step-4: free(head)
Step-5: set head=NULL and goto step 12.
Step-6: otherwise define two pointers temp1, temp2 anc
Step-7: Repeat step 8,9 while temp1->next !=head
Step-8: temp2=temp1
Step-9: temp1=temp1->next
End Loop
Step-10: set temp2->next=head
Step-11: free(temp1)
Step-12: print one node deleted.
Step-13: exit.