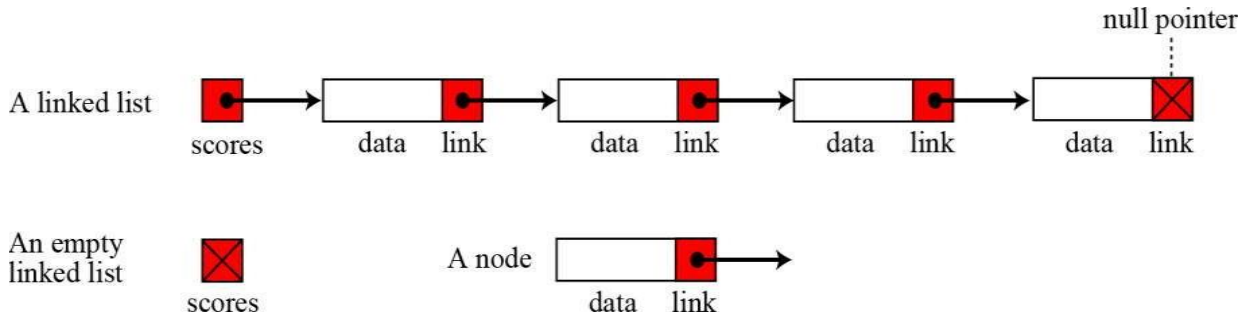
UNIT - 3

Linked Lists: Introduction, singly linked lists, circular linked lists, doubly linked lists, multiple linked lists, applications.
Linked stacks and linked queues: Introduction, operations on linked stacks and linked queues, dynamic memory management, implementation of linked representations, applications.

LINKED LIST:

Introduction:

- Linked List can be defined as 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 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 cannot 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 limited to the available memory space.

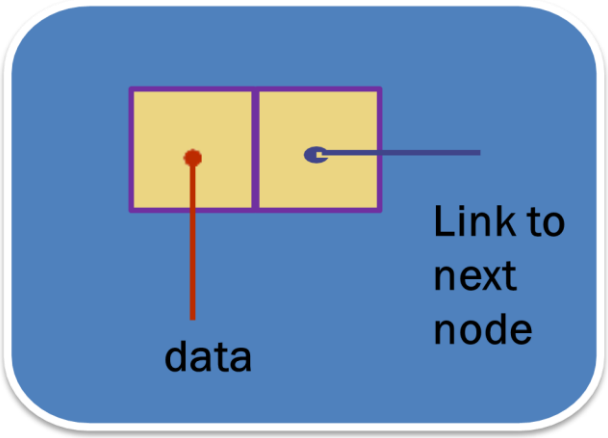
Types of linked list:

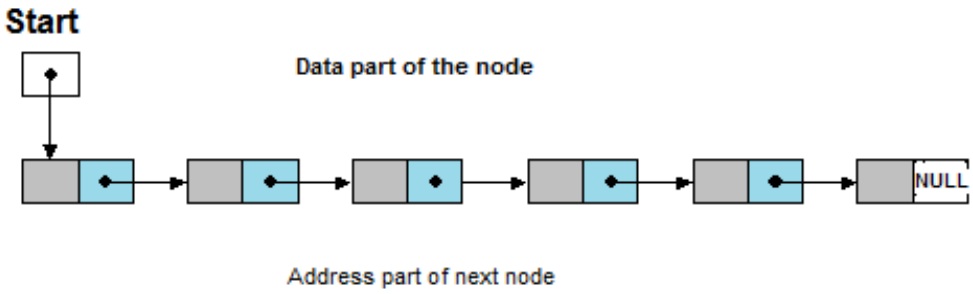
Following are the various types of linked list.

- Singly Linked List − Item navigation is forward only.
- Doubly Linked List − Items can be navigated forward and backward.
- Circular Linked List − Last item contains link of the first element as next and the first element has a link to the last element as previous.

Singly linked list:

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consists of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.





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

Operations on Singly Linked List:

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

- **Creation**
- **Insertion**
- **Deletion**
- **Traversing**

Structure of a node:

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

Creation:

- Step 1 - Include all the header files which are used in the program.
- Step 2 - Declare all the user defined functions.
- Step 3 - Define a Node pointer 'head' and set it to NULL.

```
struct Node
{
    int data;
   struct Node *next;
}*head=NULL;
```

- Step 4 - Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion:

1. **Insertion at beginning:**

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

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

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

newnode → data = value;

- 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.

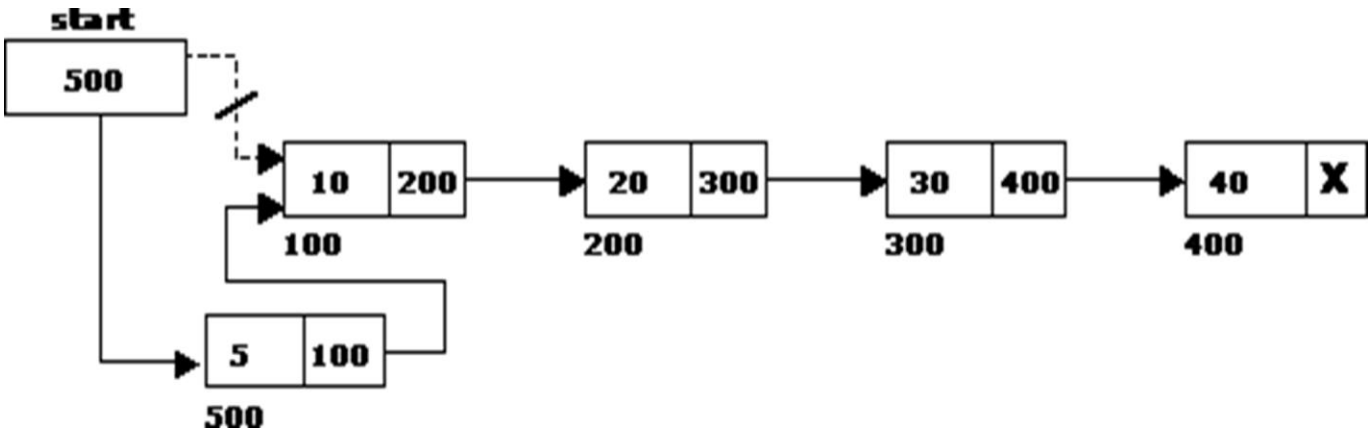
newnode → next = head;

- 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 = newnode;

Algorithm:

- **Step 1:** if newnode = null
  write overflow go to step 7
  [end of if]
- **Step 2:** create a node called newnode = (struct node*) malloc(sizeof(struct node))
- **Step 3:** set newnode → data =val
- **Step 4:** set newnode →next=head
- **Step 5:** set head = new_node
- **Step 6:** exit



2. **Insertion at specified position:**
   - In order to insert an element at specified position in 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.

```
temp = head
for(i=0;i<pos-1;i++)
{
        temp=temp→ next;
        if (temp==Null)
                break;
}
```

   - 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.

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

newnode → data = value;

   - 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.

newnode → next= temp→next

temp→next=newnode

Algorithm:

Step1: create two pointers *newnode, *temp = *head

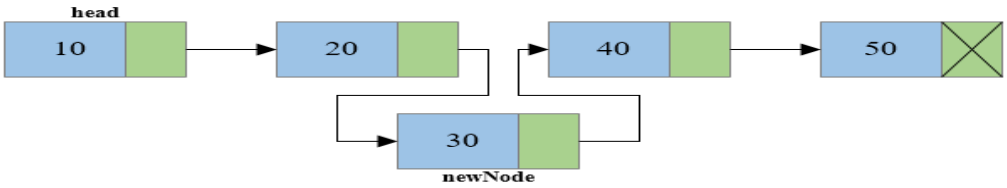
Step2: take two variable i and position

Step3: if newnode == NULL then print message
Step4: otherwise allocate a memory to newnode, assign values into it

newnode → data = value;

step5: traverse the list till specified position occur then insert the newnode with following statements.

newnode → next= temp→next
temp→next=newnode



3. **Insertion at last :**

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.

  newnode->data = item;
  newnode -> next = NULL;
- Since, **newnode** 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 = newnode

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.

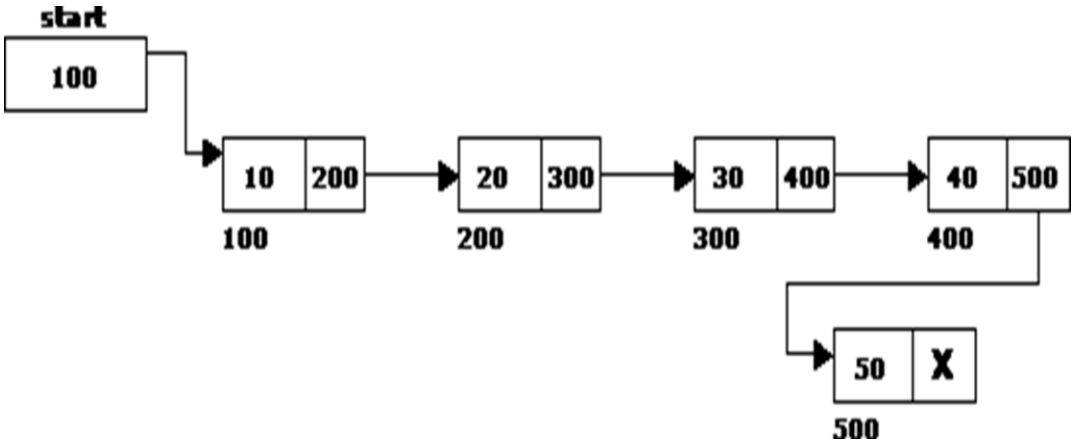
  temp = head
- Then, traverse through the entire linked list using the statements:

  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 .

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

Algorithm

- **Step 1:** IF NEWNODE = 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:** IF HEAD = NULL THEN SET HEAD=NEWNODE
- **Step 5:** OTHERWISE SET TEMP = HEAD
- **Step 6:** Repeat Step 7 while TEMP - > NEXT != NULL
- **Step 7:** SET TEMP = TEMP - > NEXT
  [END OF LOOP]
- **Step 8:** SET TEMP - > NEXT = NEWNODE
- **Step 9:** EXIT



Deletion:

1. **Deletion 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.

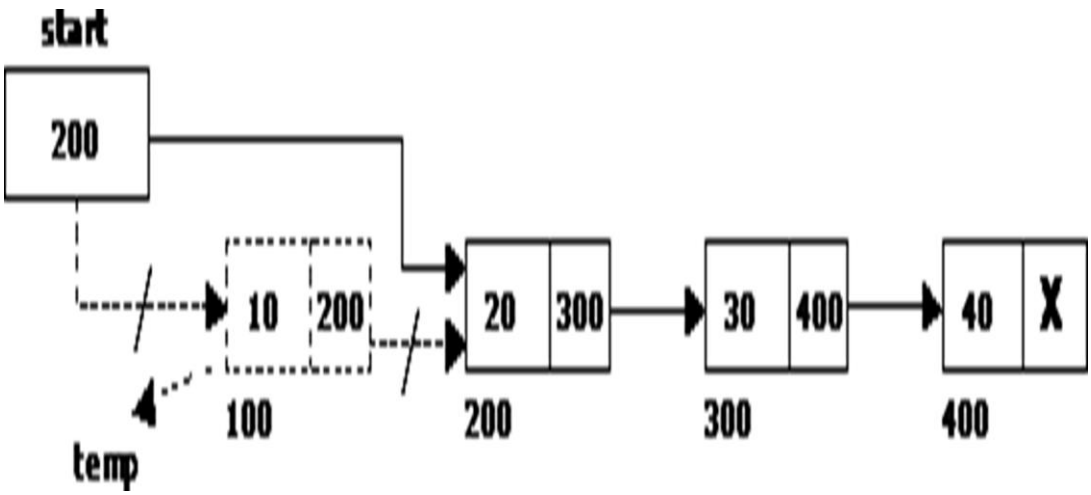
   free(ptr)

Algorithm

Step 1: IF HEAD = NULL

Write UNDERFLOW
Go to Step 5
[END OF IF]

- **Step 2:** SET PTR = HEAD
- **Step 3:** SET HEAD = HEAD -> NEXT
- **Step 4:** FREE PTR
- **Step 5:** EXIT



2. **Deletion at specified position**

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;
  }
}
```

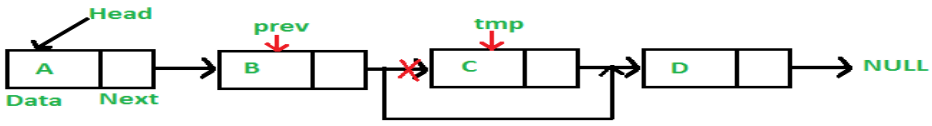
Now, our task is almost done, we just need to make a few pointer adjustments. Make the next of ptr1 (points to the specified node) point to the next of ptr (the node which is to be deleted). This will be done by using the following statements.

```
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

- **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



3. Deletion at last

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.

.

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

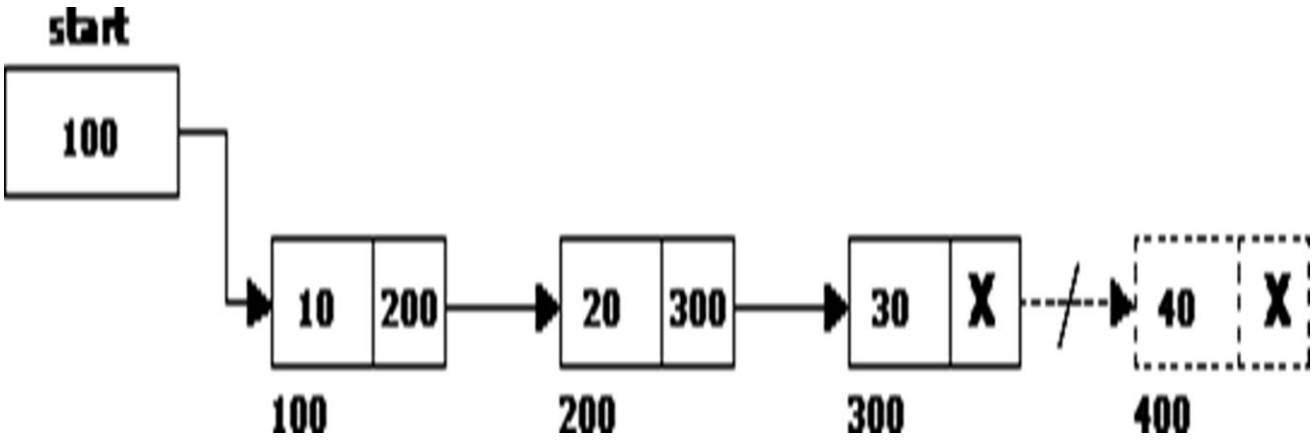
Now, we just need to make the pointer ptr1 point to the NULL and the last node of the list that is pointed by ptr will become free. It will be done by using the following statements.

```
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



Traversing:

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;
      }
```

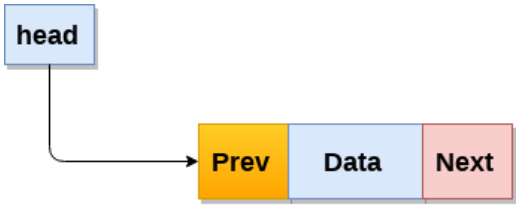
Algorithm

- **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

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

Structure of a node:

```
Struct node
{
        Int data;
        Struct node *prev;
        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.

Operations on doubly linked list:

1. Insertion
2. Deletion
3. Traversing/display
4. Searching

Insertion:

1. Insertion 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;
```

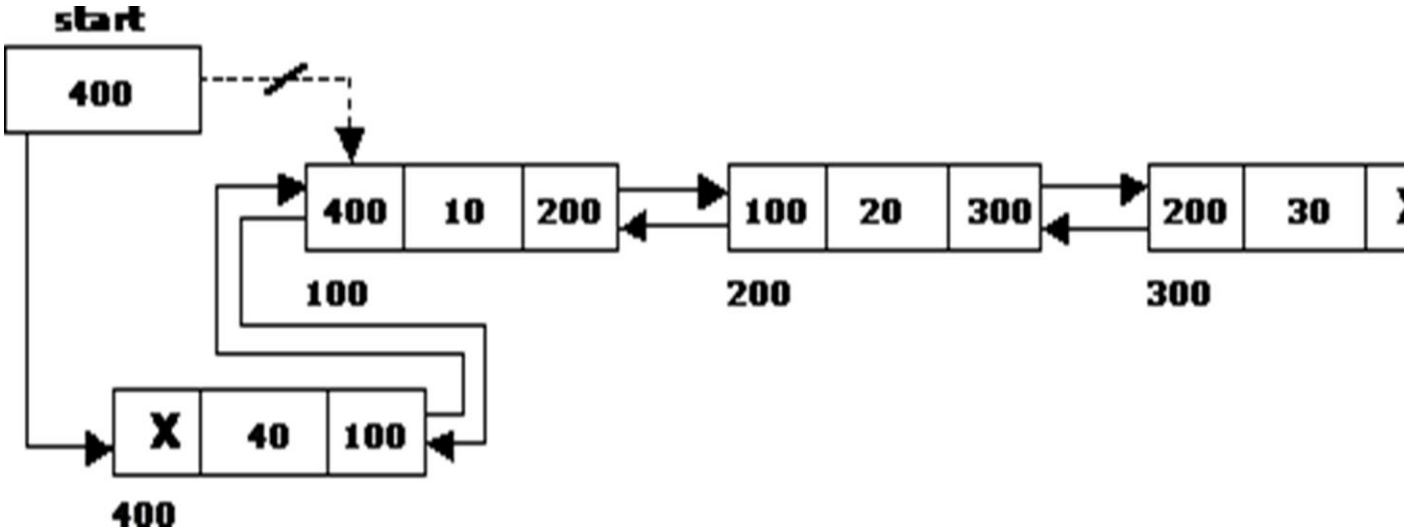
Since, the node being inserted is the first node of the list and therefore it must contain NULL in its prev pointer. Hence assign null to its previous part and make the head point to this node.

```
ptr→prev =NULL
head = ptr
```

Algorithm :

- **Step 1:** IF ptr = NULL
  Write OVERFLOW
  Go to Step 9
  [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 -> PREV = NULL

- **Step 6:** SET NEW_NODE -> NEXT = START
- **Step 7:** SET head -> PREV = NEW_NODE
- **Step 8:** SET head = NEW_NODE
- **Step 9:** EXIT



2. **Insertion at specified position:**
   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 {
        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;

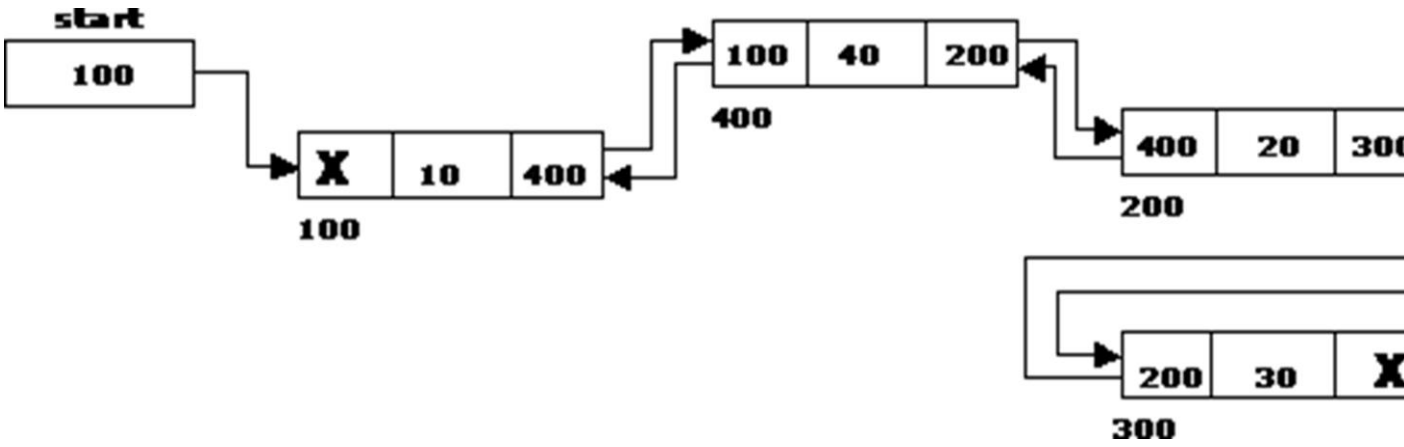
   make the **previous** pointer of the next node of temp point to the new node.
     temp → next → prev = ptr;

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
- **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



3. Insertion at last:

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.

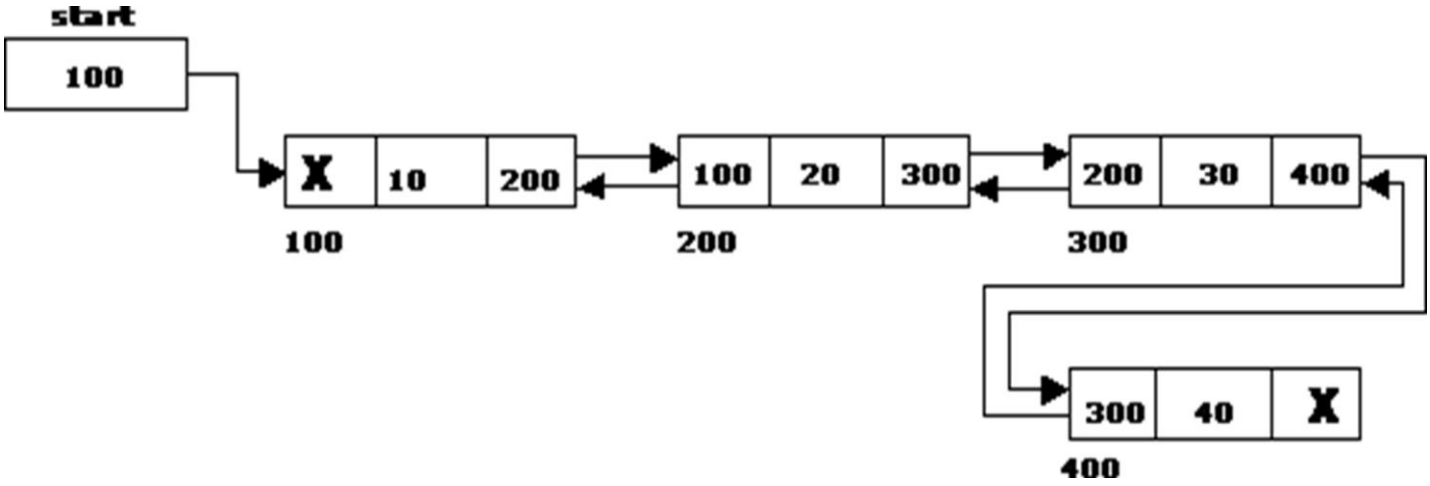
```
ptr → prev = temp;
```

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

- **Step 1:** IF PTR = NULL
  Write OVERFLOW
  Go to Step 11
  [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 = NULL
- **Step 6:** SET TEMP = START
- **Step 7:** Repeat Step 8 while TEMP -> NEXT != NULL
- **Step 8:** SET TEMP = TEMP -> NEXT
  [END OF LOOP]
- **Step 9:** SET TEMP -> NEXT = NEW_NODE
- **Step 10C:** SET NEW_NODE -> PREV = TEMP
- **Step 11:** EXIT



Deletion:

1. **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.

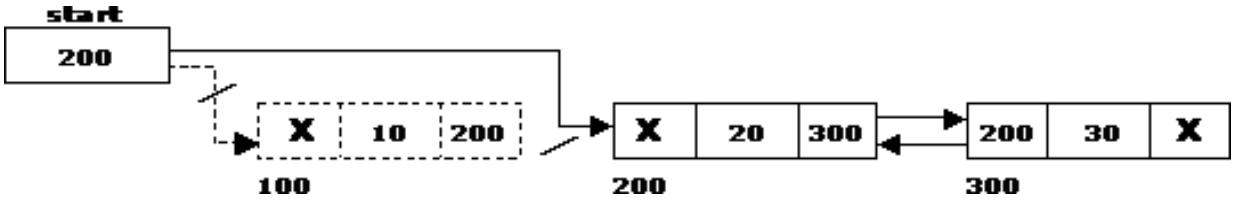
   head → prev = NULL

   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



2. **Deletion at specified position**

   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.

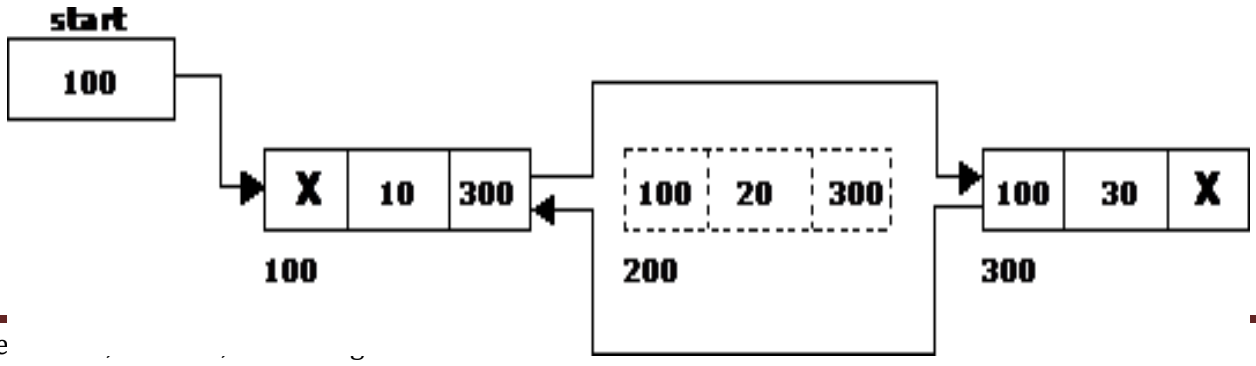
```
if(temp -> next -> next == NULL)
   {
      temp ->next = NULL;
   }
```

- 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

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



3. **Deletion at last:**

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**.

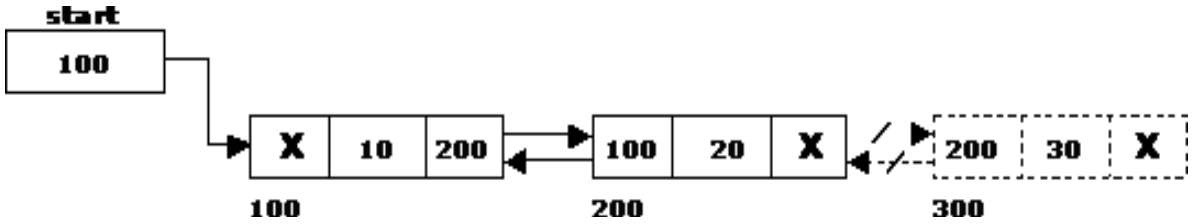
```
ptr → prev → next = NULL
```

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

```
free(ptr)
```

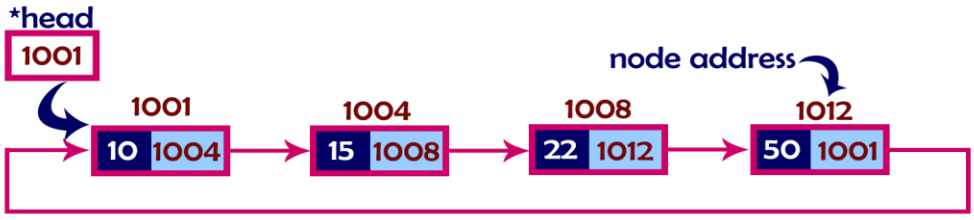
Algorithm:

Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 7
[END OF IF]
Step 2: SET TEMP = HEAD
Step 3: REPEAT STEP 4 WHILE TEMP->NEXT != NULL
Step 4: SET TEMP = TEMP->NEXT
[END OF LOOP]
Step 5: SET TEMP ->PREV-> NEXT = NULL
Step 6: FREE TEMP
Step 7: EXIT



Circular linked list:

In single linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list. **A circular linked list is a sequence of elements in which every element has a link to its next element in the sequence and the last element has a link to the first element.** That means circular linked list is similar to the single linked list except that the last node points to the first node in the list.



Operations

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

1. Insertion
2. Deletion
3. Display

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

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

Insertion

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

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

Inserting At Beginning of the list

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

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

- **Step 6 -** Set '**newNode → next** =**head**', '**head** = **newNode**' and '**temp → next** = **head**'.

Inserting At End of the list

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

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

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

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

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode → next** = **head**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- **Step 7 -** If **temp** is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).
- **Step 8 -** If **temp** is last node then set **temp → next** = **newNode** and **newNode → next** = **head**.
- **Step 8 -** If **temp** is not last node then set **newNode → next** = **temp → next** and **temp → next** = **newNode**.

Deletion

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

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

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

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

Deleting from End of the list

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

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

Deleting a Specific Node from the list

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

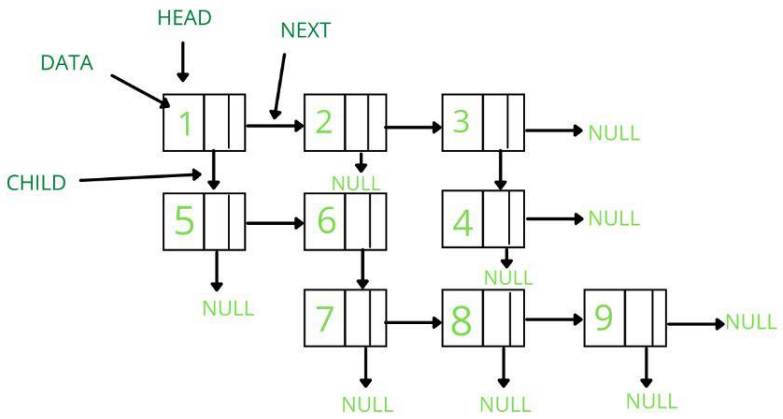
- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4 -** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
- **Step 5 -** If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
- **Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1 → next** == **head**)
- **Step 7 -** If list has only one node and that is the node to be deleted then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
- **Step 8 -** If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).
- **Step 9 -** If **temp1** is the first node then set **temp2** = **head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next**, **temp2 → nex**t = **head** and delete **temp1**.
- **Step 10 -** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).
- **Step 1 1-** If **temp1** is last node then set **temp2 → next** = **head** and delete **temp1** (**free(temp1)**).
- **Step 12 -** If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

Displaying a circular Linked List

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

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

Multiple linked list:
Multilevel Linked List is a 2D data structure that comprises several linked lists and each node in a multilevel linked list has a next and child pointer. All the elements are linked using pointers.



Representation:
A multilevel linked list is represented by a pointer to the first node of the linked lists. Similar to the linked list, the first node is called the head. If the multilevel linked list is empty, then the value of head is NULL. Each node in a list consists of at least three parts: **Data, Pointer to the next node, pointer to the child node.**

- The advantage of using the multilinked list is that the **same set of data can be processed into multiple sequences**.
- In a multilinked list, **the data are not duplicated anywhere in the list**.
- **The data is always one of a kind and exists only once in the list**, but multiple paths connect the one set of data.

Insertion

In the diagram above, **there can be seen two logical lists**. So when adding a node to the list, **we must insert it into each of the logical lists there**.

- **For different logical lists with different logical keys, we always use separate algorithms for each node insertion and search**.
- The might be situations where **we only call insert as long as we read data, otherwise, we don't call insert in case of any error or we have reached the end of the file**.
- Similarly, **we can only call insert as long as the memory allocation in the build node module is active**, otherwise, we can't call insert if it is not active or fails.

Deletion

In the node deleting operation in multilinked list, we must always reconnect the pointer for each logical list.
In the above president list example, **if we delete a president's record, we also need to adjust the spouse's as well as the president's successor pointer**

Applications of singly linked list:

1. It is used to implement **stacks** and **queues** which are like fundamental needs throughout computer science.
2. To prevent the collision between the data in the **hash map**, we use a singly linked list.

3. If we ever noticed the functioning of a casual notepad, it also uses a singly linked list to perform undo or redo or deleting functions.

4. We can think of its use in a photo viewer for having look at photos continuously in a slide show.

5. In the system of train, the idea is like a singly linked list, as if you want to add a Boggie, either you have to take a new boggie to add at last or you must spot a place in between boggies and add it.

6. Implementation of graphs : Adjacency list representation of graphs is most popular which is uses linked list to store adjacent vertices.
7. Dynamic memory allocation : We use linked list of free blocks.
8. Representation of sparse matrices

Applications of circular linked list:

1. It can also be used to implement queues by maintaining a pointer to the last inserted node and the front can always be obtained as next of last.

2. Circular Doubly Linked Lists are used for the implementation of advanced data structures like Fibonacci Heap. Also reference to the previous node can easily be found in this.

3. It is also used by the Operating system to share time for different users, generally uses a Round-Robin time-sharing mechanism (this algorithm comes from the round-robin principle, where each person gets an equal share of something in turns. It is the oldest, simplest scheduling algorithm, which is mostly used for multitasking).

4. Multiplayer games use a circular list to swap between players in a loop.

5. In photoshop, word, or any paint we use this concept in undo function.

Applications of doubly linked list:

1. Great use of the doubly linked list is in navigation systems, as it needs front and back navigation.

2. In the browser when we want to use the back or next function to change the tab, it used the concept of a doubly-linked list here.

3. Again, it is easily possible to implement other data structures like a binary tree, hash tables, stack, etc.

4. It is used in music playing system where you can easily play the previous one or next one song as many times one person wants to. Basically it provides full flexbility to perform functions and make the system user-friendly.

5. In many operating systems, the thread scheduler (the thing that chooses what processes need to run at which times) maintains a doubly-linked list of all the processes running at any time. This makes it easy to move a process from one queue (say, the list of active processes that need a turn to run) into another queue (say, the list of processes that are blocked and waiting for something to release them).

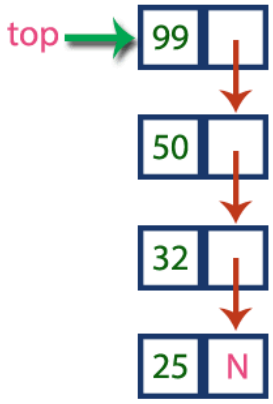
6. It is used in a famous game concept which is a deck of cards.

Linked stacks and linked queues

Stack using linked list:

The major problem with the stack implemented using an array is, it works only for a fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using an array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using a linked list data structure. The stack implemented using linked list can work for an unlimited number of values. That means, stack implemented using linked list works for the variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want. In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list. The **next** field of the first element must be always **NULL**.

- A stack using a linked list is just a simple linked list with just restrictions that any element will be added and removed using push and pop respectively. In addition to that, we also keep *top* pointer to represent the top of the stack. This is described in the picture given below.



Operations on stacks using linked list:

1. push()
2. pop()
3. display()

Push(value) - Inserting an element into the Stack.

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

Pop() - Deleting an Element from a Stack

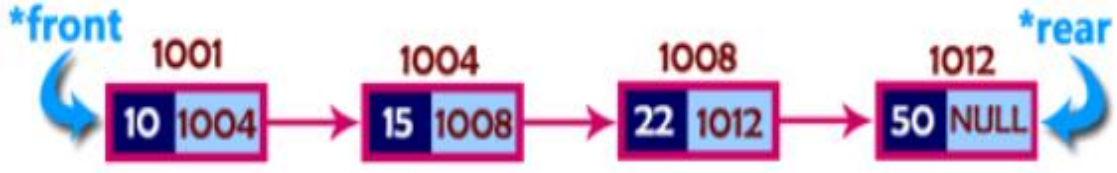
- **Step 1 -** Check whether **stack** is **Empty** (**top == NULL**).
- **Step 2 -** If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function
- **Step 3 -** If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.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

Display() - Displaying stack of elements

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

Queue using Linked List:

- A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values.
- That means, queue using linked list can work for the variable size of data. The Queue implemented using linked list can organize as many data values as we want.
- In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.
- In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.



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

Operations on Queue using Linked list:

Algorithm for create() an empty queue:

- **Step 1 -** Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2 -** Define a '**Node**' structure with two members **data** and **next**.
- **Step 3 -** Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.
- **Step 4 -** Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the **main** method to perform user selected operation.

Algorithm for Enqueue() operation:

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

Algorithm for Dequeue() operation:

- **Step 1 -** Check whether **queue** is **Empty** (**front == NULL**).

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

Algorithm for Display() operation:

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

Dynamic memory management:

Dynamic memory management refers to the process of allocating memory to the variables during execution of the program ot at run time. This allows you to obtain more memory when required and release it when not necessary. There are 4 library functions defined under <stdlib.h> for dynamic memory allocation.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer to the first byte of allocated space
calloc()	Allocates space for an array of elements, initializes them to zero and then returns a pointer to memory
free()	deallocate the previously allocated space
realloc()	Change the size of previously allocated space

malloc()

- The name malloc stands for "memory allocation".
- The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

Syntax:

ptr = (cast-type*) malloc(byte-size);

Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

Example:

ptr = (int*) malloc(100 * sizeof(int));

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

Program: to find sum of n elements entered by user using malloc() and free()

```
#include    <stdio.h>
#include    <stdlib.h>
int main()

{

   int  *ptr, n,i;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   ptr = (int*) malloc(n * sizeof(int));

   if(ptr == NULL)

   {

       printf("Memory not allocated.");exit(0);

   }

   printf("Memory allocated ");

   for(i = 0; i < n; ++i)

   {

     ptr[i]=i+1;
```

calloc()

- The name calloc stands for "contiguous allocation".
- The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax:

ptr = (cast-type*)calloc(n, element-size);

This statement will allocate contiguous space in memory for an array of n elements.

Example:

ptr = (float*) calloc(25, sizeof(float));

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

Program: to find sum of n elements entered by user using calloc() and free

```
#include    <stdio.h>
#include    <stdlib.h>
int main()

{

   int  *ptr, n,i;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   ptr = (int*) calloc(n, sizeof(int));

   if(ptr == NULL)

   {

       printf("Memory not allocated.");exit(0);

   }

   printf("Memory allocated ");

   for(i = 0; i < n; ++i)

   {

     ptr[i]=i+1;
```

free()

- Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. You must explicitly use free() to release the space.

Syntax:

free(ptr);

This statement frees the space allocated in the memory pointed by ptr.

realloc()

If the previously allocated memory is insufficient or more than required, you can change the previously allocated memory size using realloc().

Syntax:

ptr = realloc(ptr, newsize);

Here, ptr is reallocated with size of newsize.

Program

```
#include     <stdio.h>
#include    <stdlib.h>
int main()

{

    int  *ptr, n,i;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    ptr = (int*) calloc(n, sizeof(int));

    if(ptr == NULL)

    {

        printf("Memory not allocated.");exit(0);

    }

    printf("Memory allocated ");

    for(i = 0; i < n; ++i)

    {

      ptr[i]=i+1;

      printf(" %d", ptr[i]);

    }

    printf("Enter number of elements: ");
    scanf("%d", &n);

    ptr = (int*) realloc(ptr, n*sizeof(int));
```