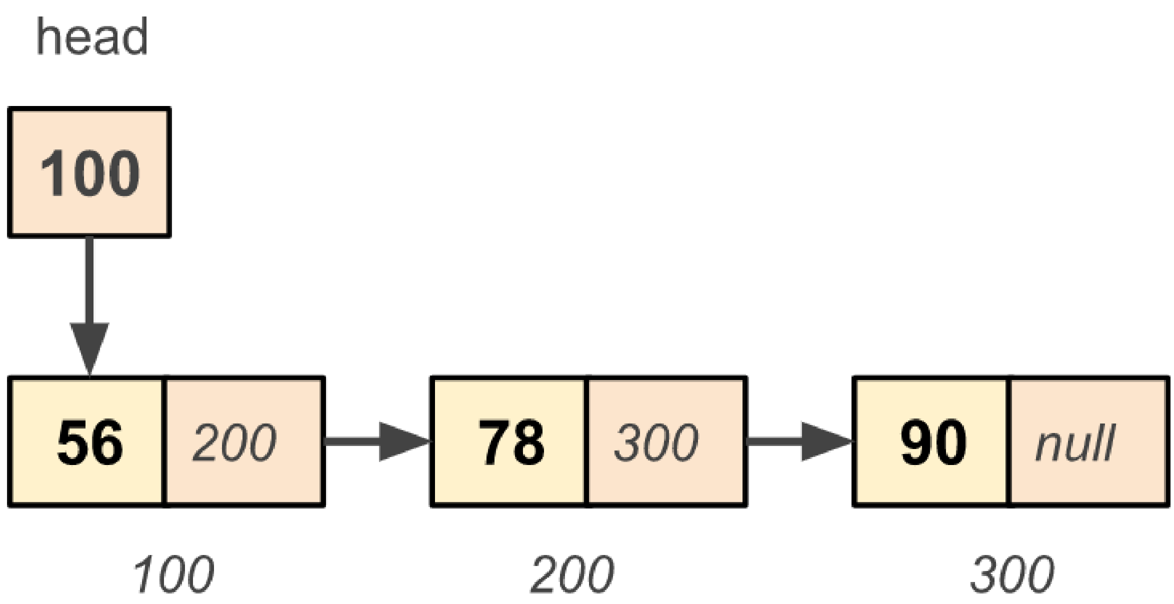
Linked List

- What is Linked List?
  The linked list is a linear data structure that contains a sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called "Node".

Types of Linked list:

The following are the types of linked list:

- Singly Linked list
- Doubly Linked list
- Circular Linked list
- SINGLE LINKED LIST:
  Single linked list is a sequence of elements in which every element has link to its next element in the sequence.



Operations on Single Linked List

The following operations are performed on a Single Linked List

- **Insertion**
- **Deletion**
- **Display**

Before we implement actual operations, first we need to set up an 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 single 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 single 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 **newNode→next** = **NULL** and **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, set **newNode→next** = **head** and **head** = **newNode**.

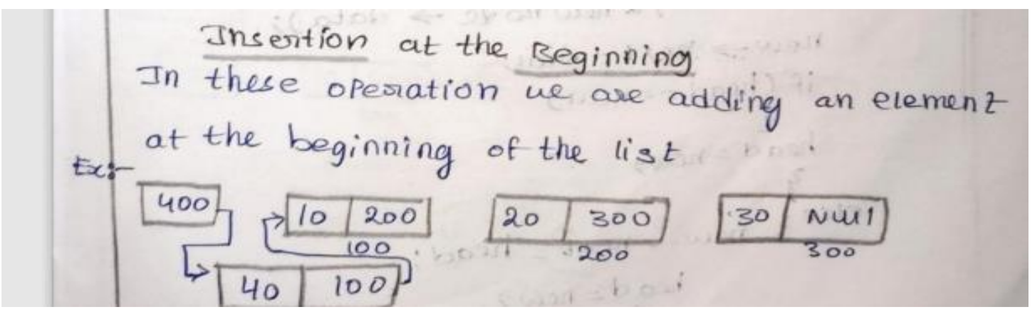
ALGORITHM:

Operations on Linked-Lists

Code to Insert at the Beginning of the Linked List

```
void insertatbeg()

{

struct node *NewNode;

int item;

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

if(NewNode == NULL)

{

printf("\nOVERFLOW");

}

else

{

printf("\nEnter value\n");

scanf("%d",&item);
```

```
NewNode->data = item;

NewNode->next = start;

start = NewNode;

printf("\nNode inserted");

}

}
```



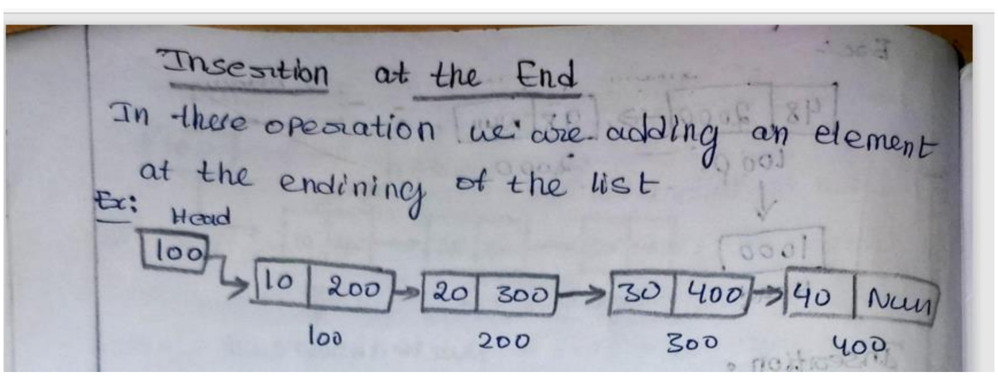
Inserting At End of the list

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

- **Step 1 -** Create a **newNode** with given value and **newNode → next** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**).
- **Step 3 -** If it is **Empty** then, set **head** = **newNode**.
- **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** is equal to **NULL**).
- **Step 6 -** Set **temp → next** = **newNode**.

ALGORITHM:

```
struct node *NewNode;

NewNode=malloc(sizeof(struct node));

NewNode-> data = 40;

NewNode->next = NULL;

struct node *temp = start;

while( temp->next ! = NULL){

temp=temp -> next;

}

temp -> next = NewNode;
```



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

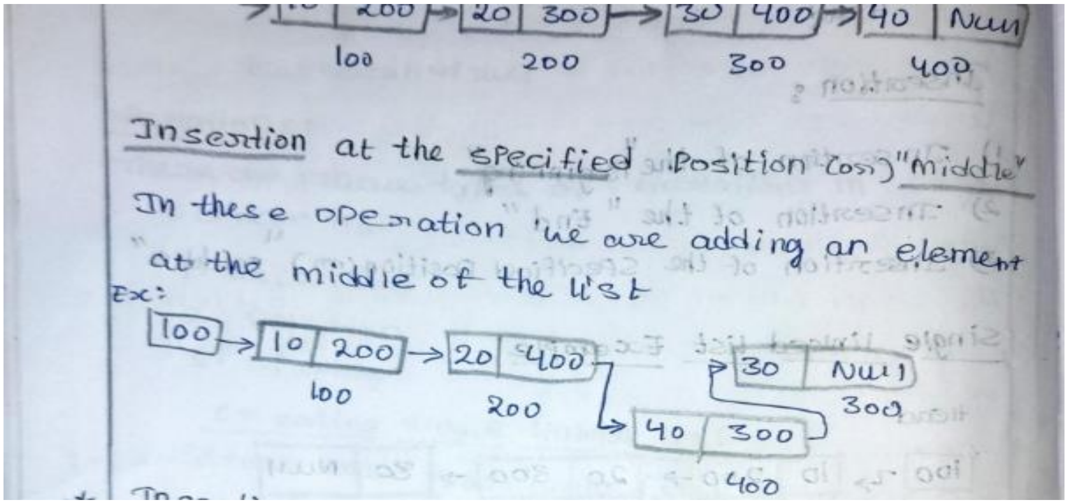
We can use the following steps to insert a new node after a node in the single linked list...

- **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 **newNode** → **next** = **NULL** and **head** = **newNode**.
- **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 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 -** Finally, Set '**newNode → next** = **temp → next**' and '**temp → next** = **newNode**'.

ALGORITHM:

```
struct node *NewNode;

NewNode= malloc(sizeof(struct node));

NewNode -> data = 40;

struct node - > temp = start;

for(int i=2; i<position; i++){

if (temp -> next!= NULL)

temp = temp -> next;

}}

NewNode -> next = temp -> next;

temp -> next = NewNode;
```



Deletion

In a single linked list, the deletion operation can be performed in three ways. They 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

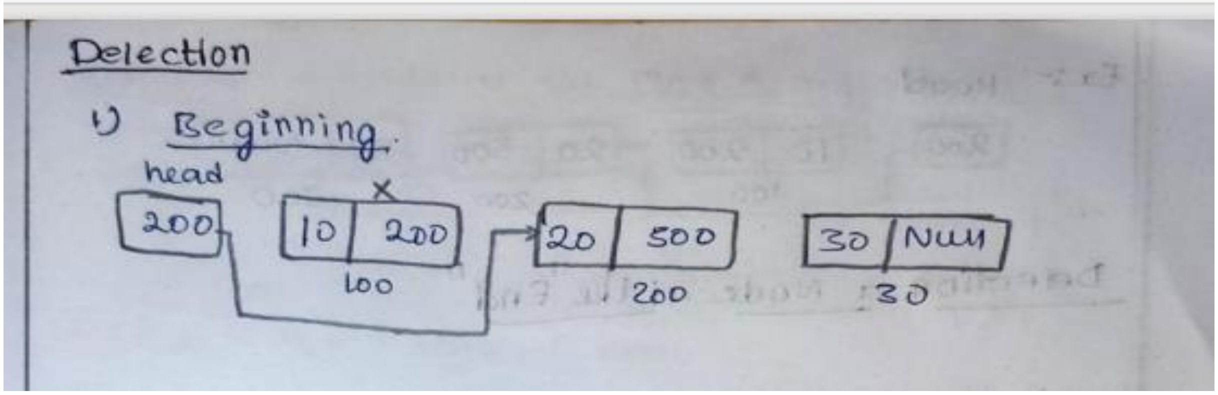
We can use the following steps to delete a node from beginning of the single linked list...

- **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 a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Check whether list is having only one node (**temp** → **next** == **NULL**)
- **Step 5 -** If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE** then
- set **head** = **temp** → **next**, and delete **temp**.

  ALGORITHM:

Code to Delete From the Beginning of the Linked List

```
void deleteatbeg()
{
struct node *NewNode;
if(start == NULL)
{
printf("\nList is empty\n");
}
else
{
NewNode = start;
start = NewNode->next;
free(NewNode);
printf("\nNode deleted from the beginning\n");
}
}
```



Deleting from End of the list:

We can use the following steps to delete a node from end of the single 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** == **NULL**)
- **Step 5 -** If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate 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 it reaches to the last node in the list. (until **temp1** → **next** == **NULL**)
- **Step 7 -** Finally, Set **temp2** → **next** = **NULL** and delete **temp1**.

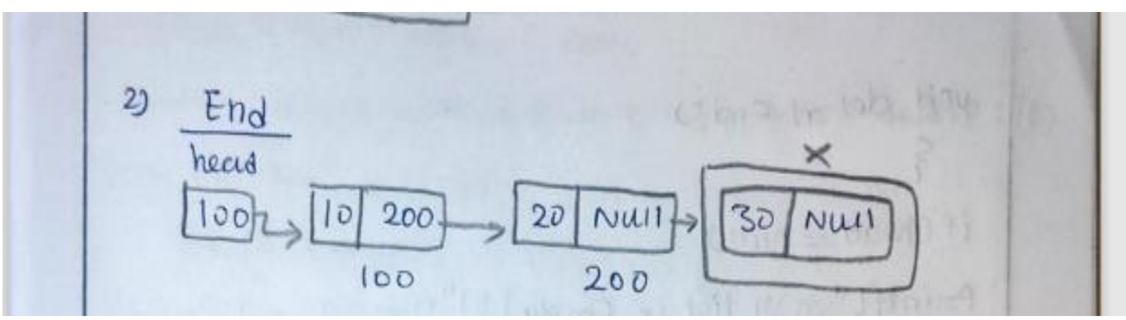
ALGORITHM:

Code to Delete From the End of the Linked List

```
void deleteatend()
```

```
{
struct node *NewNode,*NewNode1;
if(start == NULL)
{
printf("\list is empty");
}
else if(start -> next == NULL)
{
start = NULL;
free(NewNode);
printf("\n node of the list deleted\n");
}
else
{
NewNode = start;
```

```
while(NewNode->next != NULL)

{

NewNode1 = NewNode;

NewNode = NewNode ->next;

}

NewNode1->next = NULL;

free(NewNode);

printf("\nDeleted Node from the last\n");

}

}
```



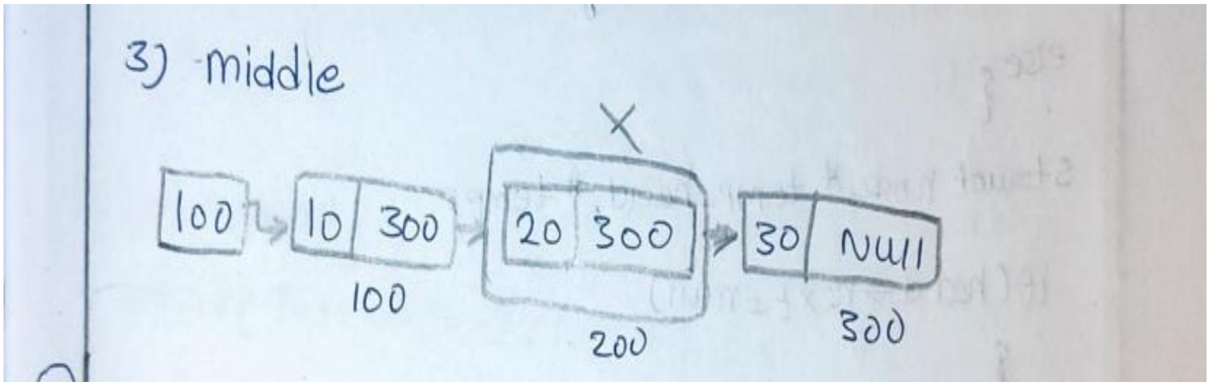
Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single 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 or not
- **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 move the **head** to the next node (**head = head → next**) and delete **temp1**.
- **Step 10 -** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == NULL**).
- **Step 11 -** If **temp1** is last node then set **temp2 → next** = **NULL** 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)**).
- **ALGORITHM:**

```
void deletemiddle()

{

struct node *NewNode,*NewNode1;

int position,i;

printf("\n what location of the node \n");
```

```
scanf("%d",&position);
NewNode=start;
for(i=0;i<position;i++)
{
NewNode1 = NewNode;
NewNode = NewNode->next;
if(NewNode == NULL)
{
printf("\nCan't delete");
return;
}
}
NewNode1 ->next = NewNode ->next;
free(NewNode);
printf("\nDeleted node %d ",position+1);
}
```



Displaying a Single Linked List

We can use the following steps to display the elements of a single 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 **NULL** (**temp** → **data** **---> NULL**).

  ALGORITHM:

```
void display()
{
   if(head == NULL)
   {
      printf("\nList is Empty\n");
   }
   else
   {
      struct Node *temp = head;
      printf("\n\nList elements are - \n");
      while(temp->next != NULL)
      {
         printf("%d --->",temp->data);
```

```
            temp = temp->next;
        }
        printf("%d --->NULL",temp->data);
    }
}
```

Circular Linked List

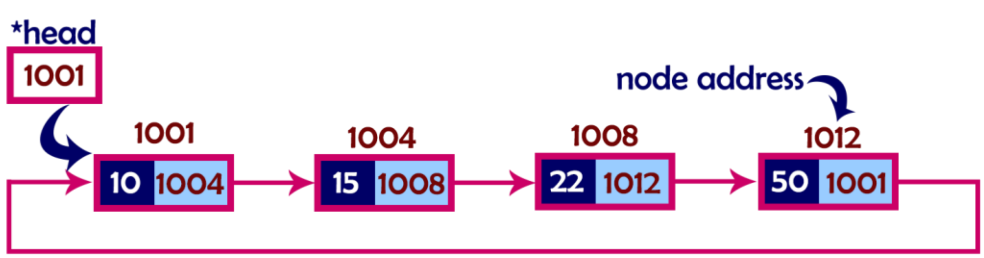
What is Circular Linked List?

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

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

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

Example



Operations

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

1. Insertion
2. Deletion
3. Display

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

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

Insertion

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

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

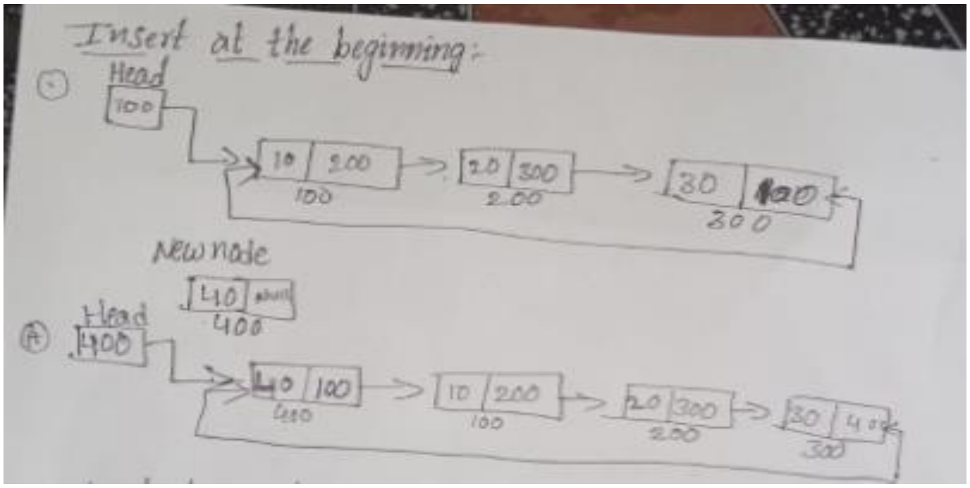
Inserting At Beginning of the list

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

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

ALGORITHM:

```
void insertAtBeginning(int value)
{
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode -> data = value;
    if(head == NULL)
    {
       head = newNode;
       newNode -> next = head;
    }
    else
    {
       struct Node *temp = head;
       while(temp -> next != head)
          temp = temp -> next;
       newNode -> next = head;
       head = newNode;
       temp -> next = head;
    }
    printf("\nInsertion success!!!");
}
```



Inserting At End of the list

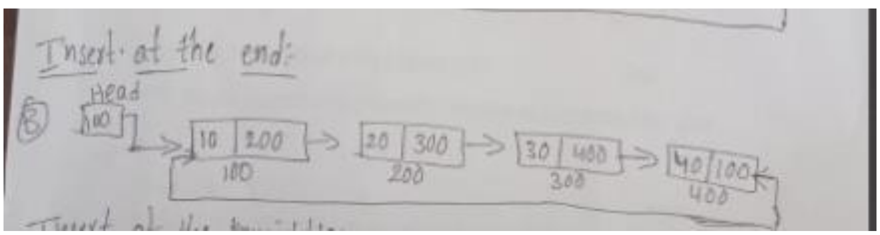
We can use the following steps to insert a new node at end of the circular linked list...

- **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**.

ALGORTHIM:

```
void insertAtEnd(int value)

{

   struct Node *newNode;

   newNode = (struct Node*)malloc(sizeof(struct Node));

   newNode -> data = value;

   if(head == NULL)

   {
```

```
        head = newNode;
        newNode -> next = head;
    }
    else
    {
        struct Node *temp = head;
        while(temp -> next != head)
            temp = temp -> next;
        temp -> next = newNode;
        newNode -> next = head;
    }
    printf("\nInsertion success!!!");
}
```



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...

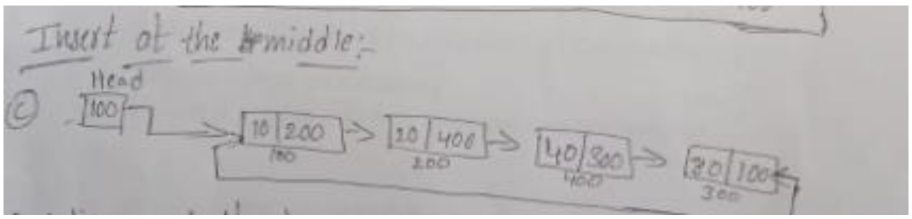
- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode** → **next** = **head**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp**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

ALGORTHIM:

```
void insertAfter(int value, int location)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode -> data = value;
   if(head == NULL)
   {
      head = newNode;
      newNode -> next = head;
   }
   else
   {
      struct Node *temp = head;
      while(temp -> data != location)
      {
         if(temp -> next == head)
         {
            printf("Given node is not found in the list!!!");

         }
         else
         {
            temp = temp -> next;
         }
      }
```

```
        newNode -> next = temp -> next;
        temp -> next = newNode;
        printf("\nInsertion success!!!");
    }
}
```



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

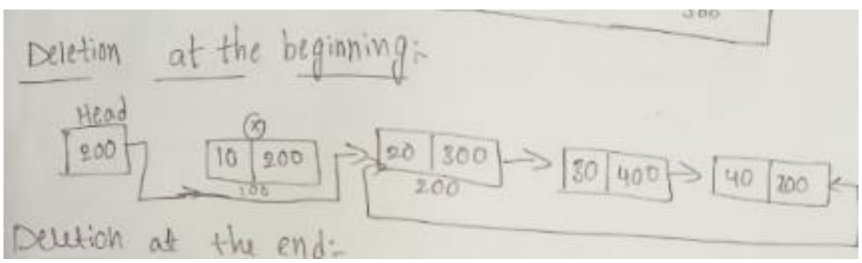
Deleting from Beginning of the list

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**.

ALGORITHM:

```
void deleteBeginning()
{
   if(head == NULL)
      printf("List is Empty!!! Deletion not possible!!!");
   else
   {
      struct Node *temp = head;
      if(temp -> next == head)
      {
         head = NULL;
         free(temp);
      }
      else{
         head = head -> next;
         free(temp);
      }
      printf("\nDeletion success!!!");
   }

}
```



Deleting from End of the list

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

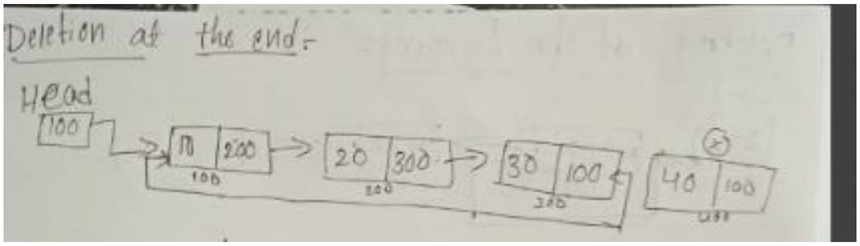
- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)

- **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**.

ALGORITHM:

```
void deleteEnd()
{
   if(head == NULL)
      printf("List is Empty!!! Deletion not possible!!!");
   else
   {
      struct Node *temp1 = head, temp2;
      if(temp1 -> next == head)
      {
         head = NULL;
         free(temp1);
      }
      else{
         while(temp1 -> next != head){
            temp2 = temp1;
            temp1 = temp1 -> next;
         }
         temp2 -> next = head;
         free(temp1);
      }
      printf("\nDeletion success!!!");
   }
```

}



Deleting a Specific Node from the list

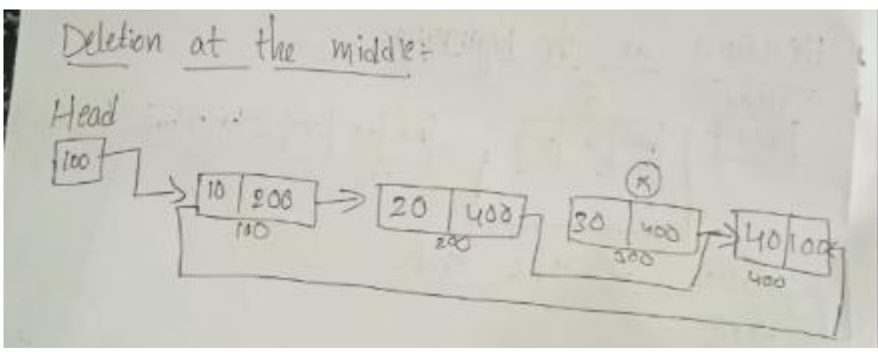
We can use the following steps to delete a specific node from the circular linked list...

- **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)**).

ALGORITHM:

```
}
void deleteSpecific(int delValue)
{
   if(head == NULL)
      printf("List is Empty!!! Deletion not possible!!!");
   else
   {
      struct Node *temp1 = head, temp2;
      while(temp1 -> data != delValue)
      {
         if(temp1 -> next == head)
         {
            printf("\nGiven node is not found in the list!!!");
         }
         else
         {
            temp2 = temp1;
            temp1 = temp1 -> next;
         }
      }
      if(temp1 -> next == head){
         head = NULL;
         free(temp1);
      }
      else{
         if(temp1 == head)
         {
            temp2 = head;
            while(temp2 -> next != head)
               temp2 = temp2 -> next;
            head = head -> next;
            temp2 -> next = head;
            free(temp1);
         }
```

```
            else
            {
               if(temp1 -> next == head)
               {
                  temp2 -> next = head;
               }
               else
               {
                  temp2 -> next = temp1 -> next;
               }
               free(temp1);
            }
         }
         printf("\nDeletion success!!!");
      }
   }
```



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**.

ALGORITHM:

```
void display()
{
   if(head == NULL)
      printf("\nList is Empty!!!");
   else
   {
      struct Node *temp = head;
      printf("\nList elements are: \n");
      while(temp -> next != head)
      {
         printf("%d ---> ",temp -> data);
      }
      printf("%d ---> %d", temp -> data, head -> data);
   }
}
```

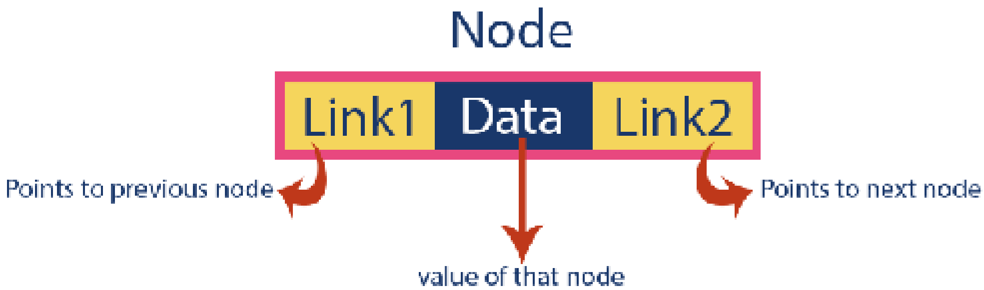
Double Linked List

What is Double Linked List?

In a single linked list, every node has a link to its next node in the sequence. So, we can traverse from one node to another node only in one direction and we can not traverse back. We can solve this kind of problem by using a double linked list. A double linked list can be defined as follows...

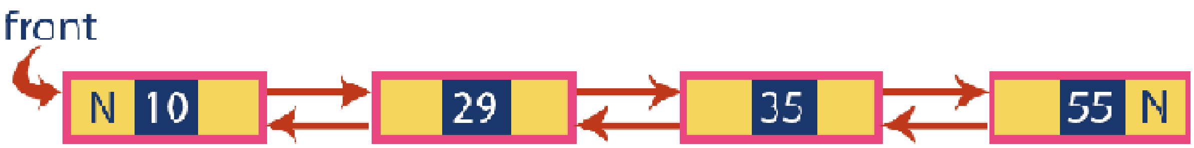
Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

In a double linked list, every node has a link to its previous node and next node. So, we can traverse forward by using the next field and can traverse backward by using the previous field. Every node in a double linked list contains three fields and they are shown in the following figure...



Here, **'link1'** field is used to store the address of the previous node in the sequence, **'link2'** field is used to store the address of the next node in the sequence and **'data'** field is used to store the actual value of that node.

Example



Operations on Double Linked List

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

1. Insertion
2. Deletion
3. Display

Insertion

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

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

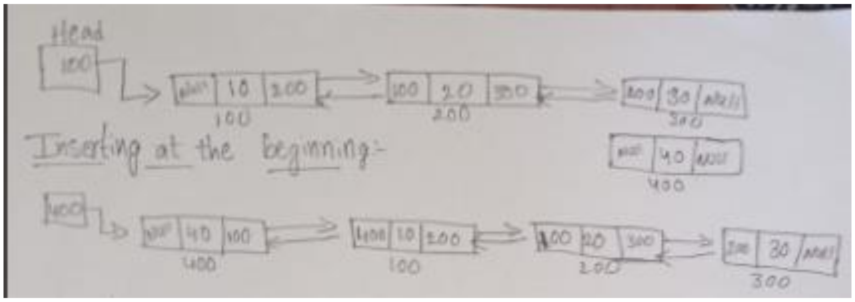
Inserting At Beginning of the list

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

- **Step 1 -** Create a **newNode** with given value and **newNode** → **previous** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, assign **NULL** to **newNode** → **next** and **newNode** to **head**.
- **Step 4 -** If it is **not Empty** then, assign **head** to **newNode** → **next** and **newNode** to **head**.

ALGORITHM:

```
void insertAtBeginning(int value)
{
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode -> data = value;
    newNode -> previous = NULL;
    if(head == NULL)
    {
       newNode -> next = NULL;
       head = newNode;
    }
    else
    {
       newNode -> next = head;
       head = newNode;
    }
    printf("\nInsertion success!!!");
}
```



Inserting At End of the list

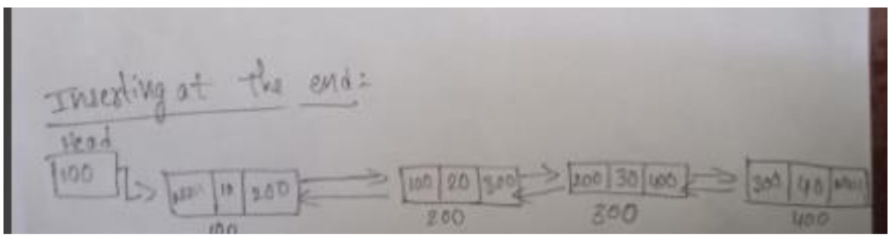
We can use the following steps to insert a new node at end of the double linked list...

- **Step 1 -** Create a **newNode** with given value and **newNode → next** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty**, then assign **NULL** to **newNode → previous** and **newNode** to **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** is equal to **NULL**).
- **Step 6 -** Assign **newNode** to **temp → next** and **temp** to **newNode → previous**.

ALGORITHM:

```
void insertAtEnd(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode -> data = value;
   newNode -> next = NULL;
   if(head == NULL)
   {
      newNode -> previous = NULL;
      head = newNode;
   }
```

```
    else
    {
       struct Node *temp = head;
       while(temp -> next != NULL)
          temp = temp -> next;
       temp -> next = newNode;
       newNode -> previous = temp;
    }
    printf("\nInsertion success!!!");
 }
```



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 double linked list...

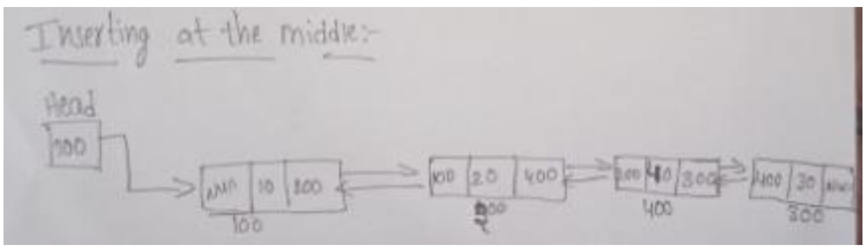
- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, assign **NULL** to both **newNode → previous** & **newNode → next** and set **newNode** to **head**.
- **Step 4 -** If it is **not Empty** then, define two node pointers **temp1** & **temp2** and initialize **temp1** with **head**.

- **Step 5 -** Keep moving the **temp1** 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 **temp1** is reached to the last node. If it is reached to the last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp1** to next node.
- **Step 7 -** Assign **temp1** → **next** to **temp2**, **newNode** to **temp1** → **next**, **temp1** to **newNode** → **previous**, **temp2** to **newNode** → **next** and **newNode** to **temp2** → **previous**.

ALGORITHM:

```
void insertAfter(int value, int location)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode -> data = value;
   if(head == NULL)
   {
      newNode -> previous = newNode -> next = NULL;
      head = newNode;
   }
   else
   {
      struct Node *temp1 = head, temp2;
      while(temp1 -> data != location)
      {
         if(temp1 -> next == NULL)
         {
            printf("Given node is not found in the list!!!");
          }
         else
```

```
            {
               temp1 = temp1 -> next;
            }
         }
         temp2 = temp1 -> next;
         temp1 -> next = newNode;
         newNode -> previous = temp1;
         newNode -> next = temp2;
         temp2 -> previous = newNode;
         printf("\nInsertion success!!!");
      }
   }
```



Deletion

In a double linked list, the deletion operation can be performed in three ways 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

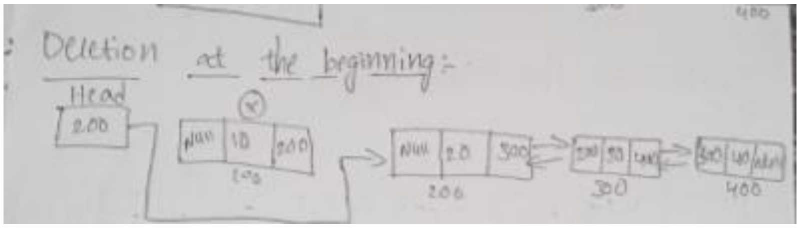
We can use the following steps to delete a node from beginning of the double linked list...

- **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 a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Check whether list is having only one node (**temp** → **previous** is equal to **temp** → **next**)
- **Step 5 -** If it is **TRUE**, then set **head** to **NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE**, then assign **temp** → **next** to **head**, **NULL** to **head** → **previous** and delete **temp**.

ALGORITHM:

```
void deleteBeginning()
{
   if(head == NULL)
      printf("List is Empty!!! Deletion not possible!!!");
   else
   {
      struct Node *temp = head;
      if(temp -> previous == temp -> next)
      {
         head = NULL;
         free(temp);
      }
      else{
         head = temp -> next;
         head -> previous = NULL;
         free(temp);
      }
      printf("\nDeletion success!!!");
   }
}
```



Deleting from End of the list

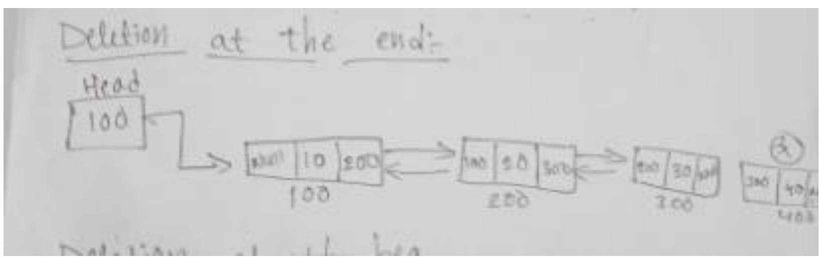
We can use the following steps to delete a node from end of the double linked list...

- **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 a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Check whether list has only one Node (**temp** → **previous** and **temp** → **next** both are **NULL**)
- **Step 5 -** If it is **TRUE**, then assign **NULL** to **head** and delete **temp**. And terminate from the function. (Setting **Empty** list condition)
- **Step 6 -** If it is **FALSE**, then keep moving **temp** until it reaches to the last node in the list. (until **temp** → **next** is equal to **NULL**)
- **Step 7 -** Assign **NULL** to **temp** → **previous** → **next** and delete **temp**.

ALGORITHM:

```
}
void deleteEnd()
{
   if(head == NULL)
      printf("List is Empty!!! Deletion not possible!!!");
```

```
    else
    {
       struct Node *temp = head;
       if(temp -> previous == temp -> next)
       {
          head = NULL;
          free(temp);
       }
       else{
          while(temp -> next != NULL)
             temp = temp -> next;
          temp -> previous -> next = NULL;
          free(temp);
       }
       printf("\nDeletion success!!!");
    }
 }
```



Deleting a Specific Node from the list

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

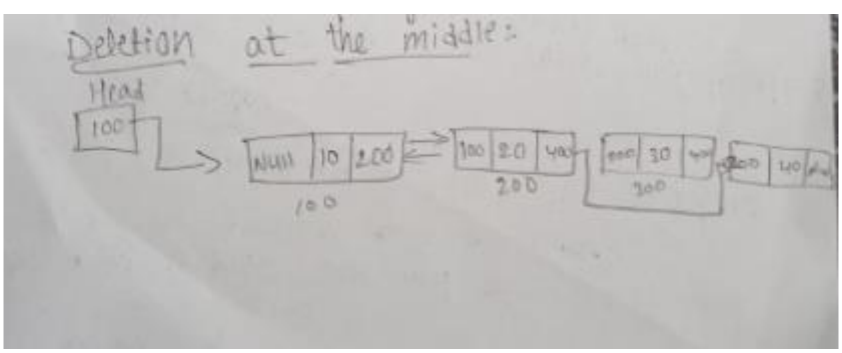
- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

- **Step 3 -** If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Keep moving the **temp** until it reaches to the exact node to be deleted or to the last 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 fuction.
- **Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- **Step 7 -** If list has only one node and that is the node which is to be deleted then set **head** to **NULL** and delete **temp** (**free(temp)**).
- **Step 8 -** If list contains multiple nodes, then check whether **temp** is the first node in the list (**temp == head**).
- **Step 9 -** If **temp** is the first node, then move the **head** to the next node (**head = head** → **next**), set **head** of **previous** to **NULL** (**head** → **previous = NULL**) and delete **temp**.
- **Step 10 -** If **temp** is not the first node, then check whether it is the last node in the list (**temp** → **next == NULL**).
- **Step 11 -** If **temp** is the last node then set **temp** of **previous** of **next** to **NULL** (**temp** → **previous** → **next = NULL**) and delete **temp** (**free(temp)**).
- **Step 12 -** If **temp** is not the first node and not the last node, then set **temp** of **previous** of **next** to **temp** of **next** (**temp** → **previous** → **next** = **temp** → **next**), **temp** of **next** of **previous** to **temp** of **previous** (**temp** → **next** → **previous** = **temp** → **previous**) and delete **temp** (**free(temp)**).

ALGORITHM:

```
void deleteSpecific(int delValue)
{
   if(head == NULL)
      printf("List is Empty!!! Deletion not possible!!!");
   else
```

```
    {
       struct Node *temp = head;
       while(temp -> data != delValue)
       {
          if(temp -> next == NULL)
          {
             printf("\nGiven node is not found in the list!!!");
          }
          else
          {
             temp = temp -> next;
          }
       }
       if(temp == head)
       {
          head = NULL;
          free(temp);
       }
       else
       {
          temp -> previous -> next = temp -> next;
          free(temp);
       }
       printf("\nDeletion success!!!");
    }
}
```



Displaying a Double Linked List

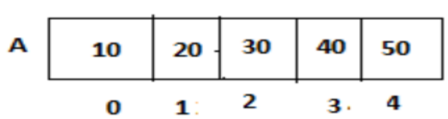
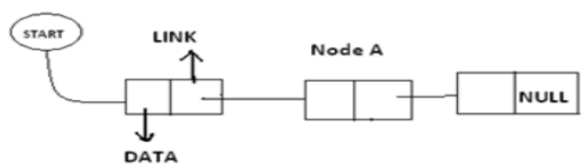
We can use the following steps to display the elements of a double linked list...

- **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 -** Display **'NULL <--- '**.
- **Step 5 -** Keep displaying **temp** → **data** with an arrow (**<===>**) until **temp** reaches to the last node
- **Step 6 -** Finally, display **temp** → **data** with arrow pointing to **NULL** (**temp** → **data** **---> NULL**).

ALGORITHM:

```
void display()
{
   if(head == NULL)
      printf("\nList is Empty!!!");
   else
   {
      struct Node *temp = head;
      printf("\nList elements are: \n");
      printf("NULL <--- ");
      while(temp -> next != NULL)
      {
         printf("%d <===> ",temp -> data);
      }
      printf("%d ---> NULL", temp -> data);
   }
}
```

Difference between Arrays and Linked List?

Arrays	Linked List
1. Arrays are used in the predictable storage requirement ie; exert amount of data storage required by the program can be determined.	1. Linked List are used in the unpredictable storage requirement ie; exert amount of data storage required by the program can't be determined.
2. In arrays the operations such as insertion and deletion are done in an inefficient manner.	2. In Linked List the operations such as insertion and deletion are done more efficient manner ie; only by changing the pointer.
3. The insertion and deletion are done by moving the elements either up or down.	3. The insertion and deletion are done by only changing the pointers.
4. Successive elements occupy adjacent space on memory.	4. Successive elements need not occupy adjacent space.
5. In arrays each location contain DATA only 6. The linear relation ship between the data elements of an array is reflected by the physical relation ship of data in the memory.	5. In linked list each location contains data and pointer to denote whether the next element present in the memory.
7. In array declaration a block of memory space is required.	6. The linear relation ship between the data elements of a Linked List is reflected by the Linked field of the node.
8.There is no need of storage of pointer or lines	7. In Linked list there is no need of such thing. 8. In Linked list a pointer is stored along into the element.
9.The Conceptual view of an Array is as follows:  	9. The Conceptual view of Linked list is as follows: 
10.In array there is no need for an element to specify whether the next is stored	10. There is need for an element (node) to specify whether the next node is formed.

6. Explain in detail about polynomials using singly linked lists. 10M

Ans:

A polynomial p(x) is the expression in variable x which is in the form ($ax^n + bx^{n-1} + \ldots + jx + k$), where a, b, c …., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

- one is the coefficient
- other is the exponent

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

Points to keep in Mind while working with Polynomials:

- The sign of each coefficient and exponent is stored within the coefficient and the exponent itself
- Additional terms having equal exponent is possible one
- The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent