Linked List :

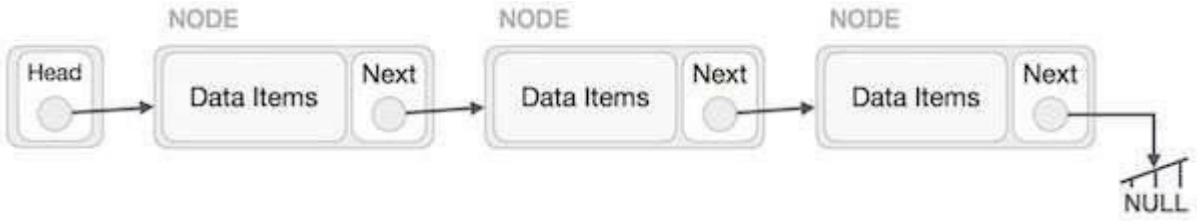
A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of nodes which contains items. Each node contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** – Each node of a linked list can store a data called an element.
- **Next** – Each node of a linked list contains a link to the next node called Next.
- **Linked List** – A Linked List contains the connection link to the first node called First.

Linked List Representation:

Linked list can be visualized as a chain of nodes, where every node points to the next node.



As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each node carries a data field(s) and a link field called next.
- Each node is linked with its next node using its next link.
- Last link carries a link as null to mark the end of the list.

Types of Linked List:

Following are the various types of linked list.

- **Simple 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.

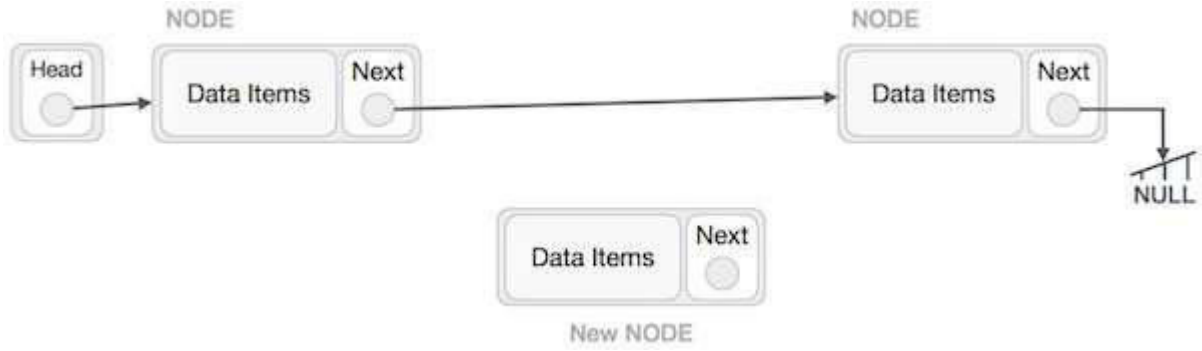
Basic Operations:

Following are the basic operations supported by a list.

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

Insertion Operation:

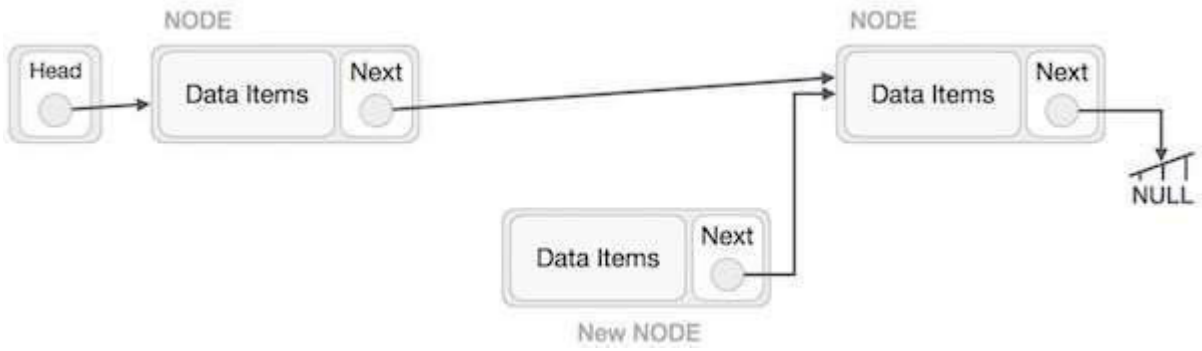
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



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

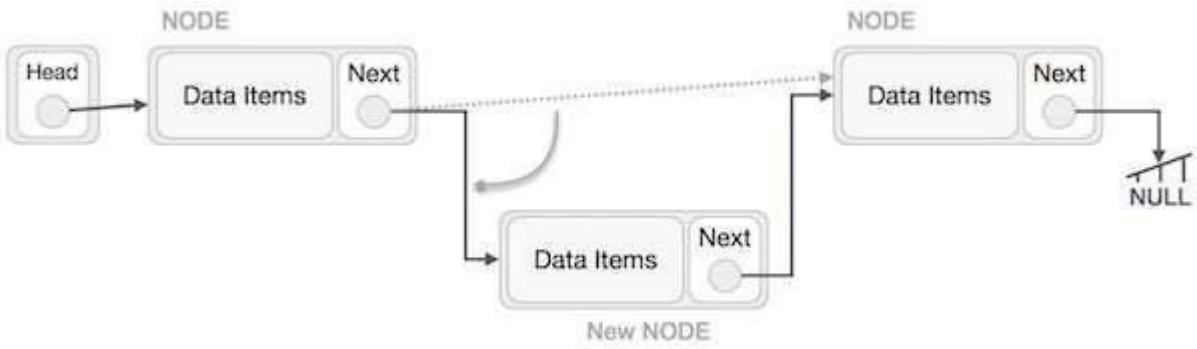
```
NewNode.next -> RightNode;
```

It should look like this –



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

```
LeftNode.next -> NewNode;
```



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

Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

Deletion Operation:

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.

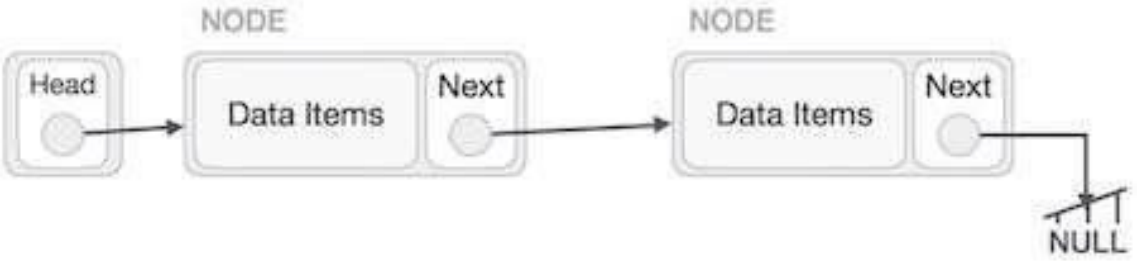
The left (previous) node of the target node now should point to the next node of the target node –

```
LeftNode.next -> TargetNode.next;
```

This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

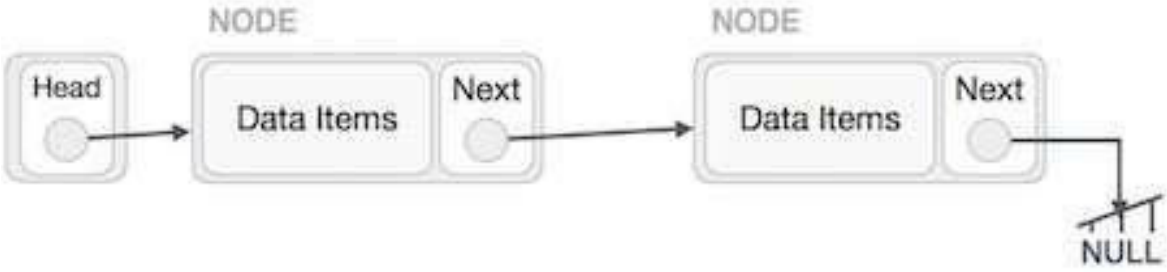
```
TargetNode.next -> NULL;
```

We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

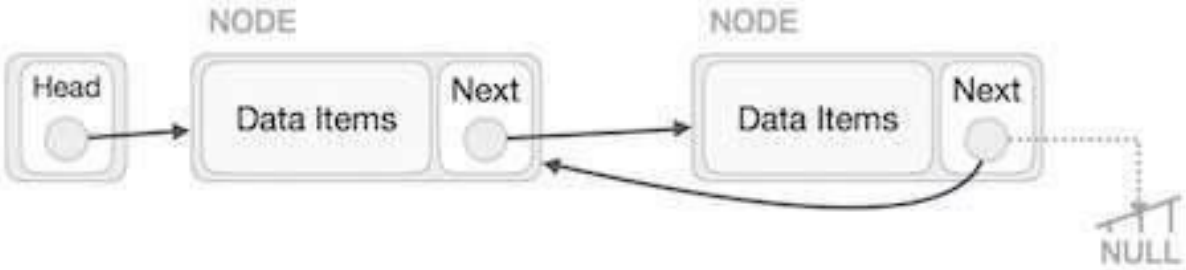


Reverse Operation:

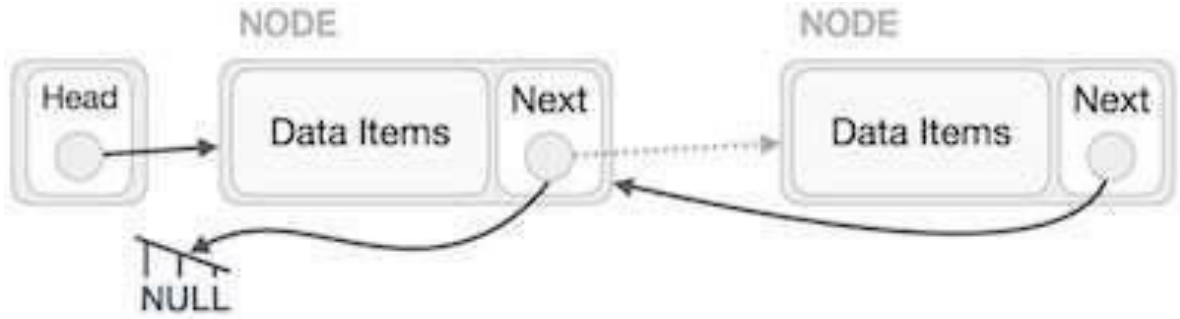
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



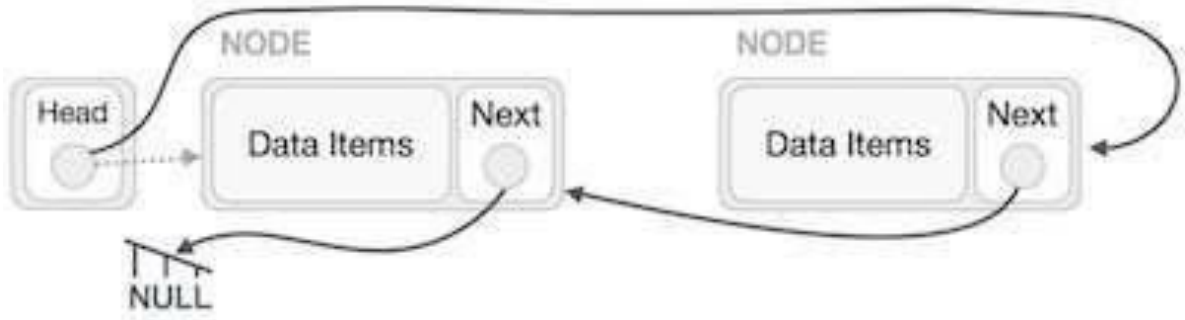
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node –



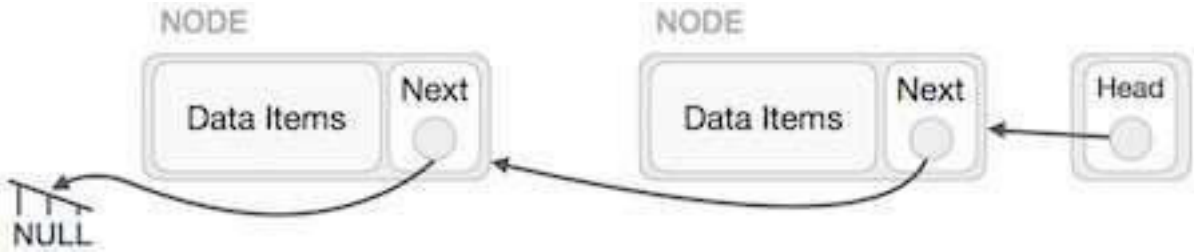
We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.



We'll make the head node point to the new first node by using the temp node.



Linked List Program in C:

A linked list is a sequence of data structures, which are connected together via links. Linked List is a sequence of nodes which contains items. Each node contains a connection to another node. Linked list is the second most-used data structure after array.

Implementation in C:

```
#include    <stdio.h>
#include   <string.h>
#include   <stdlib.h>
#include <stdbool.h>

struct node
{
   int data;
   int key;
   struct node *next;
};

struct node *head = NULL;
struct node *current = NULL;

//display the list
void printList()
{
   struct node *ptr = head;
   printf("\n[ ");

   //start from the beginning
   while(ptr != NULL)
     {
      printf("(%d,%d) ",ptr->key,ptr->data);
      ptr = ptr->next;
   }

   printf(" ]");
}

//insert link at the first location
void insertFirst(int key, int data)
{
   //create a link
   struct node *link = (struct node*) malloc(sizeof(struct node));
```

```
   link->key = key;
   link->data = data;

   //point it to old first
   node link->next = head;

   //point first to new first
   node head = link;
}

//delete first item struct
node* deleteFirst()
{

   //save reference to first link
   struct node *tempLink = head;

   //mark next to first link as
   first head = head->next;

   //return the deleted
   link return tempLink;
}

//is list empty
bool isEmpty()
{
   return head == NULL;
}

int length()
{
   int length = 0; struct
   node *current;


   for(current = head; current != NULL; current = current->next)
```

```
      {
       length++;
      }

   return length;
}

//find a link with given key
struct node* find(int key){

   //start from the first link
   struct node* current = head;

   //if list is empty
   if(head == NULL)
     {
      return NULL;
   }

   //navigate through list
   while(current->key != key){

      //if it is last node
      if(current->next == NULL){
         return NULL;
      }else {
         //go to next link current
         = current->next;
      }
   }

   //if data found, return the current
   Link return current;
}


//delete a link with given key
```

```
struct node* delete(int key){

   //start from the first link
   struct node* current = head;
   struct node* previous = NULL;

    //if list is empty
    if(head == NULL){
       return NULL;
   }

   //navigate through list
   while(current->key != key){

      //if it is last node
      if(current->next == NULL){
         return NULL;
      }else {
         //store reference to current
         link previous = current;
         //move to next link
         current = current->next;
      }

   }

   //found a match, update the link
   if(current == head) {
      //change first to point to next link
      head = head->next;
   }else {
      //bypass the current link
      previous->next = current->next;
   }
   return current;
}
void sort(){
```

```
   int i, j, k, tempKey, tempData ;
   struct node *current;
   struct node *next;

   int size = length();
   k = size ;
   for ( i = 0 ; i < size - 1 ; i++, k-- ) {
      current = head ;
      next = head->next ;

      for ( j = 1 ; j < k ; j++ ) {

         if ( current->data > next->data ) {
            tempData = current->data ;
            current->data = next->data;
            next->data = tempData ;

            tempKey = current->key;
            current->key = next->key;
            next->key = tempKey;
         }

         current = current->next;
         next = next->next;
      }
   }
}

void reverse(struct node** head_ref)
   { struct node* prev = NULL;
   struct node* current =
   *head_ref; struct node* next;
```

```
   while (current != NULL) {
      next = current->next;
      current->next = prev;
      prev = current;
      current = next;
   }
   *head_ref = prev;
}

main() {

   insertFirst(1,10);
   insertFirst(2,20);
   insertFirst(3,30);
   insertFirst(4,1);
   insertFirst(5,40);
   insertFirst(6,56);

   printf("Original List: ");

   //print list
   printList();

   while(!isEmpty()){
      struct node *temp = deleteFirst();
      printf("\nDeleted value:");
      printf("(%d,%d) ",temp->key,temp->data);
   }

   printf("\nList after deleting all items: ");
   printList();
   insertFirst(1,10);
   insertFirst(2,20);
   insertFirst(3,30);
   insertFirst(4,1);
   insertFirst(5,40);
```

```
insertFirst(6,56);
printf("\nRestored List: ");
printList();
printf("\n");

struct node *foundLink = find(4);

if(foundLink != NULL){
   printf("Element found: ");
   printf("(%d,%d) ",foundLink->key,foundLink->data);
   printf("\n");
}else {
   printf("Element not found.");
}

delete(4);
printf("List after deleting an item: ");
printList();
printf("\n");
foundLink = find(4);

if(foundLink != NULL){
   printf("Element found: ");
   printf("(%d,%d) ",foundLink->key,foundLink->data);
   printf("\n");
}else {
   printf("Element not found.");
}

printf("\n");
sort();

printf("List after sorting the data: ");
printList();
reverse(&head);
printf("\nList after reversing the data: ");
printList();
```

```
}
```

If we compile and run the above program, it will produce the following result –

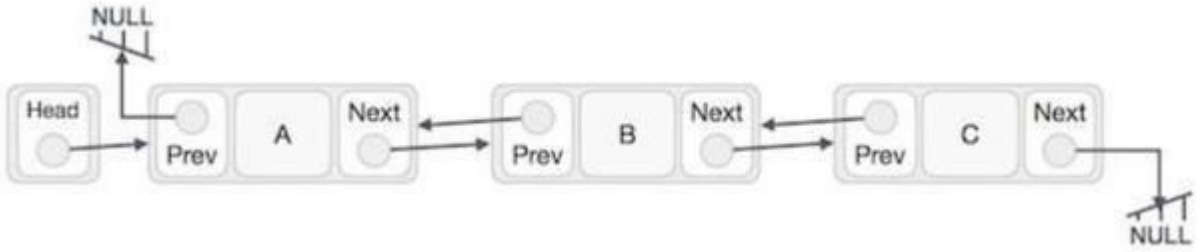
```
Original List:
[ (6,56) (5,40) (4,1) (3,30) (2,20) (1,10) ]
Deleted value:(6,56)
Deleted value:(5,40)
Deleted value:(4,1)
Deleted value:(3,30)
Deleted value:(2,20)
Deleted value:(1,10)
List after deleting all items:
[ ]
Restored List:
[ (6,56) (5,40) (4,1) (3,30) (2,20) (1,10) ]
Element found: (4,1)
List after deleting an item:
[ (6,56) (5,40) (3,30) (2,20) (1,10) ]
Element not found.
List after sorting the data:
[ (1,10) (2,20) (3,30) (5,40) (6,56) ]
List after reversing the data:
[ (6,56) (5,40) (3,30) (2,20) (1,10) ]
```

Doubly Linked List:

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

- **Link** − Each node of a linked list can store a data called an element.
- **Next** − Each node of a linked list contains a link to the next link called Next.
- **Prev** − Each node of a linked list contains a link to the previous link called Prev.
- **Linked List** − A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List Representation:



As per the above illustration, following are the important points to be considered.

- Doubly Linked List contains a link element called first and last.
- Each node carries a data field(s) and a link field called next.
- Each node is linked with its next node using its next link.
- Each node is linked with its previous node using its previous link.
- The last link carries a link as null to mark the end of the list.

Basic Operations:

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Insert Last** − Adds an element at the end of the list.
- **Delete Last** − Deletes an element from the end of the list.

- **Insert After** – Adds an element after an item of the list.
- **Delete** – Deletes an element from the list using the key.
- **Display forward** – Displays the complete list in a forward manner.
- **Display backward** – Displays the complete list in a backward manner.

Insertion Operation:

Following code demonstrates the insertion operation at the beginning of a doubly linked list.

```
//insert link at the first location
void insertFirst(int key, int data) {

   //create a link
   struct node *link = (struct node*) malloc(sizeof(struct
   node)); link->key = key;
   link->data = data;

   if(isEmpty()) {
      //make it the last link
      last = link;
   }else {
      //update first prev link
      head->prev = link;
   }

   //point it to old first
   link link->next = head;

   //point first to new first
   link head = link;
}
```

Deletion Operation:

Following code demonstrates the deletion operation at the beginning of a doubly linked list.

```
//delete first item
struct node* deleteFirst() {

   //save reference to first link
   struct node *tempLink = head;

   //if only one link
   if(head->next == NULL) {
      last =
   NULL; }else {
      head->next->prev = NULL;
   }

   head = head->next;

   //return the deleted
   link return tempLink;
}
```

Insertion at the End of an Operation:

Following code demonstrates the insertion operation at the last position of a doubly linked list.

```
//insert link at the last location
void insertLast(int key, int data) {

   //create a link
   struct node *link = (struct node*) malloc(sizeof(struct
   node)); link->key = key;
   link->data = data;
```

```
   if(isEmpty()) {
      //make it the last link
      last = link;
   }else {
      //make link a new last
      link last->next = link;
      //mark old last node as prev of new link
      link->prev = last;
   }

   //point last to new last
   node last = link;
}
```

Doubly Linked List Program in C:

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List.

Implementation in C:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

struct node {
   int data;
   int key;

   struct node *next;
   struct node *prev;
};



//this link always point to first Link
```

```
struct node *head = NULL;
//this link always point to last Link
struct node *last = NULL;

struct node *current = NULL;

//is list empty
bool isEmpty(){
   return head == NULL;
}

int length(){
   int length = 0; struct
   node *current;

   for(current = head; current != NULL; current = current->next){
      length++;
   }

   return length;
}

//display the list in from first to last
void displayForward(){

   //start from the beginning
   struct node *ptr = head;

   //navigate till the end of the
   list printf("\n[ ");

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

   printf(" ]");
```

```
}

//display the list from last to first
void displayBackward(){

   //start from the last
   struct node *ptr = last;

   //navigate till the start of the
   list printf("\n[ ");

   while(ptr != NULL){

      //print data
      printf("(%d,%d) ",ptr->key,ptr->data);

      //move to next item
      ptr = ptr ->prev;
      printf(" ");
   }

   printf(" ]");
}

//insert link at the first location
void insertFirst(int key, int data){

   //create a link
   struct node *link = (struct node*) malloc(sizeof(struct
   node)); link->key = key;
   link->data = data;

   if(isEmpty()){
      //make it the last link
      last = link;
    }else {
      //update first prev link
```

```
      head->prev = link;
   }

   //point it to old first
   link link->next = head;

   //point first to new first
   link head = link;
}

//insert link at the last location
void insertLast(int key, int data){

   //create a link
   struct node *link = (struct node*) malloc(sizeof(struct
   node)); link->key = key;
   link->data = data;

   if(isEmpty()){
      //make it the last link
      last = link;
   }else {
      //make link a new last
      link last->next = link;
      //mark old last node as prev of new link
      link->prev = last;
   }

   //point last to new last
   node last = link;
}




//delete first item
```

```
struct node* deleteFirst(){ //save
   reference to first link struct
   node *tempLink = head;

   //if only one link
   if(head->next == NULL){
      last =
   NULL; }else {
      head->next->prev = NULL;
   }

   head = head->next;
   //return the deleted
   link return tempLink;
}

//delete link at the last location

struct node* deleteLast(){ //save
   reference to last link struct
   node *tempLink = last;

   //if only one link
   if(head->next == NULL){
      head =
   NULL; }else {
      last->prev->next = NULL;
   }

   last = last->prev;

   //return the deleted
   link return tempLink;
}
```

```
//delete a link with given key

struct node* delete(int key){

   //start from the first link
   struct node* current = head;
   struct node* previous = NULL;

    //if list is empty
    if(head == NULL){
       return NULL;
   }

   //navigate through list
   while(current->key != key){
      //if it is last node

      if(current->next ==
         NULL){ return NULL;
      }else {
         //store reference to current
         link previous = current;

         //move to next link
         current = current->next;
      }
   }

   //found a match, update the
   link if(current == head) {
      //change first to point to next link
      head = head->next;
   }else {
      //bypass the current link current-
      >prev->next = current->next;
   }
   if(current == last){
```

```
      //change last to point to prev
      link last = current->prev;
   }else {
      current->next->prev = current->prev;
   }

   return current;
}

bool insertAfter(int key, int newKey, int
   data){ //start from the first link
   struct node *current = head;

    //if list is empty
    if(head == NULL){
      return false;
   }

   //navigate through list
   while(current->key != key){

      //if it is last node
      if(current->next == NULL){
         return false;
      }else {
         //move to next link
         current = current->next;
      }
   }

   //create a link
   struct node *newLink = (struct node*) malloc(sizeof(struct node));
   newLink->key = key;

   newLink->data = data;
```

```
   if(current == last) {
      newLink->next = NULL;
      last = newLink;
   }else {
      newLink->next = current->next;
      current->next->prev = newLink;
   }

   newLink->prev = current;
   current->next = newLink;
   return true;
}

main() {
   insertFirst(1,10);
   insertFirst(2,20);
   insertFirst(3,30);
   insertFirst(4,1);
   insertFirst(5,40);
   insertFirst(6,56);

   printf("\nList (First to Last):
   "); displayForward();

   printf("\n");
   printf("\nList (Last to first):
   "); displayBackward();

   printf("\nList , after deleting first record:
   "); deleteFirst();
   displayForward();

   printf("\nList , after deleting last record: ");
   deleteLast();
   displayForward();

   printf("\nList , insert after key(4) : ");
```

```
   insertAfter(4,7, 13);
   displayForward();

   printf("\nList , after delete key(4) : ");
   delete(4);
   displayForward();
}
```

If we compile and run the above program, it will produce the following result –

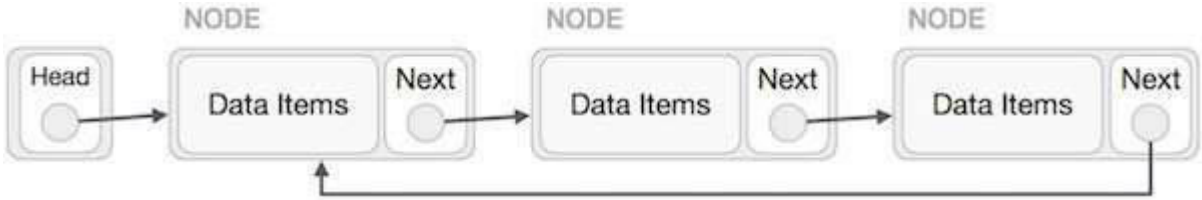
```
List (First to Last):
[ (6,56) (5,40) (4,1) (3,30) (2,20) (1,10) ]

List (Last to first):
[ (1,10) (2,20) (3,30) (4,1) (5,40) (6,56) ]
List , after deleting first record:
[ (5,40) (4,1) (3,30) (2,20) (1,10) ]
List , after deleting last record:
[ (5,40) (4,1) (3,30) (2,20) ]
List , insert after key(4) :
[ (5,40) (4,1) (4,13) (3,30) (2,20) ]
List , after delete key(4) :
[ (5,40) (4,13) (3,30) (2,20) ]
```

Circular Linked List:

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

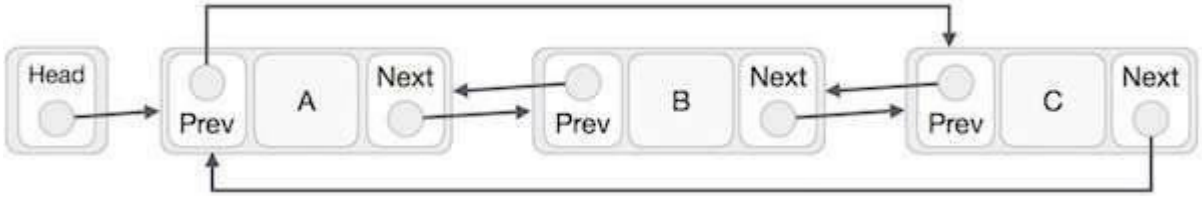
Singly Linked List as Circular:

In singly linked list, the next pointer of the last node points to the first node.



Doubly Linked List as Circular:

In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



As per the above illustration, following are the important points to be considered.

- The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
- The first link's previous points to the last of the list in case of doubly linked list.

Basic Operations:

Following are the important operations supported by a circular list.

- **insert** – Inserts an element at the start of the list.
- **delete** – Deletes an element from the start of the list.
- **display** – Displays the list.

Insertion Operation:

Following code demonstrates the insertion operation in a circular linked list based on single linked list.

```
//insert link at the first location
void insertFirst(int key, int data) {
   //create a link
   struct node  *link = (struct node*) malloc(sizeof(struct node));
   link->key =  key;
   link->data=  data;

   if (isEmpty()) { head
      = link; head->next
      = head;
   }else {
      //point it to old first node
      link->next = head;

      //point first to new first
      node head = link;
   }

}
```

Deletion Operation:

Following code demonstrates the deletion operation in a circular linked list based on single linked list.

```
//delete first item
struct node * deleteFirst() {
   //save reference to first link
   struct node *tempLink = head;

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

```
   //mark next to first link as
   first head = head->next;

   //return the deleted
   link return tempLink;
}
```

Display List Operation:

Following code demonstrates the display list operation in a circular linked list.

```
//display the list
void printList() {
   struct node *ptr = head;
   printf("\n[ ");

   //start from the beginning
   if(head != NULL) {
      while(ptr->next != ptr) {
         printf("(%d,%d) ",ptr->key,ptr->data);
         ptr = ptr->next;
      }
   }

   printf(" ]");
}
```

Circular Linked List Program in C:

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

Implementation in C:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

struct node {
   int data;
   int key;

   struct node *next;
};

struct node *head = NULL;
struct node *current = NULL;

bool isEmpty(){
   return head == NULL;
}

int length(){
   int length = 0;

   //if list is empty
   if(head == NULL){
      return 0;
   }

   current = head->next;

   while(current != head){
      length++;
      current = current->next;
   }
```

```
   return length;
}

//insert link at the first location
void insertFirst(int key, int data){

   //create a link
   struct node *link = (struct node*) malloc(sizeof(struct
   node)); link->key = key;
   link->data = data;

   if (isEmpty()) { head
      = link; head->next
      = head;
   }else {
      //point it to old first node
      link->next = head;

      //point first to new first
      node head = link;
   }

}

//delete first item
struct node * deleteFirst(){

   //save reference to first link
   struct node *tempLink = head;

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

```
   //mark next to first link as
   first head = head->next;

   //return the deleted
   link return tempLink;
}

//display the list
void printList(){

   struct node *ptr = head;
   printf("\n[ ");

   //start from the beginning
   if(head != NULL){

      while(ptr->next != ptr){
         printf("(%d,%d) ",ptr->key,ptr->data);
         ptr = ptr->next;
      }

   }

   printf(" ]");
}

main() {

   insertFirst(1,10);
   insertFirst(2,20);
   insertFirst(3,30);
   insertFirst(4,1);
   insertFirst(5,40);
   insertFirst(6,56);
```

```
   printf("Original List: ");

   //print list
   printList();

   while(!isEmpty()){
      struct node *temp = deleteFirst();
      printf("\nDeleted value:"); printf("(%d,%d)
      ",temp->key,temp->data);
   }

   printf("\nList after deleting all items: ");
   printList();
}
```

If we compile and run the above program, it will produce the following result –

```
Original List:
[ (6,56) (5,40) (4,1) (3,30) (2,20) ]
Deleted value:(6,56)
Deleted value:(5,40)
Deleted value:(4,1)
Deleted value:(3,30)
Deleted value:(2,20)
Deleted value:(1,10)
List after deleting all items: [
]
```

References:

1. “Data Structures & Algorithm” , Tutorials Point India (https://www.tutorialspoint.com)