MODULE 2: LINKED LISTS

2.1 Introduction of Linked Lists:

The linked list is very different type of collection from an array. Using such lists, we can store collections of information limited only by the total amount of memory that the OS will allow us to use. Furthermore, there is no need to specify our needs in advance. The linked list is very flexible dynamic data structure i.e., items may be added to it or deleted from it at will. A programmer need not worry about how many items a program will have to accommodate in advance. This allows us to write robust programs which require much less maintenance.

The linked allocation has the following draw backs:

- No direct access to a particular element.
- Additional memory required for pointers.

There are three types of linked list:

- Single linked list.
- Double linked list.
- Circular linked list.

2.1.1 Difference between Array and Linked Lists:

BASIS FOR	ARRAY	LINKED LIST
Basic	It is a consistent set of a fixed	It is an ordered set consisting of a
Size	Specified during declaration.	No need to specify; grow and shrink
Storage Allocation	Element location is allocated	Element position is assigned during run
Order of the	Stored consecutively	Stored randomly
Accessing the	Direct or randomly accessed, i.e.,	Sequentially accessed, i.e., Traverse
Insertion and	Slow relatively as shifting is	Easier, fast and efficient.
Searching	Binary search and linear search	linear search
Memory required	less	More
Memory Utilization	Ineffective	Efficient

2.2 Single Linked List:

A singly linked list is a linear collection of data items. The linear order is given by means of pointers. These types of lists are often referred to as linear linked list. Simply single linked list is a sequence of elements in which every element has link to its next element in the sequence.

It can be represented as follows:

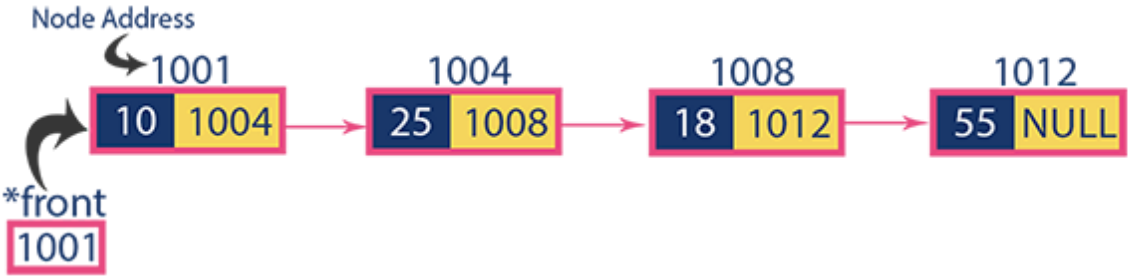


Fig 2.1: Single Linked List

Note:

1. ***In a single linked list, the address of the first node is always stored in a reference node known as "front" (It is also known as "head").***
2. ***Always next part (reference part) of the last node must be NULL.***

In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data and next. The data field is used to store actual value of that node and next field is used to store the address of the next node in the sequence.

Fig 2.2: Single node in a Single Linked List

2.2.1 Operations on Single Linked List:

Operation that we perform on a single linked list are:

- Creation of a node
- Insertion
- Deletion
- Traversing the list

2.2.1.1 Creation of a node:

A node in a single linked list consists of two fields i.e., data and link (refer fig. 2.2). In the data field we store the data that is given by the user and in the link field we store the address of the next node to be accessed, in this way we create a link between 2 nodes and further so we create a linked list.

We know that data structures are used to store and organize data in a particular way. In single linked list we want to store data (can be of any datatype such as int, float, char, etc) in a node and organize them by linking the nodes together to form a structure as shown in Figure 2.1. So, for that we first need to create nodes with 2 fields i.e., *data* and *link* (or *next*) as shown in Figure 2.2.

```
1. struct LinkedList {
2. int data;
3. struct LinkedList * next;
4. };
```

Code Snippet 2.1: Data Structure for a Node in Single Linked List

For creating a node, we need to first initialize the data structure that can be used again and again later in the program anytime we need to create a node. In C language it can be implemented by using structures and pointers. In Code Snippet 2.1, we have created a structure named *LinkedList* which consists of two fields i.e., *data* & *next* (or link). This definition is used to create every node in the list. The data field stores the element and the next is a pointer to store the address of the next node.

In place of a data type, struct LinkedList is written before next. That's because it's a self-referencing pointer. It means a pointer that points to whatever it is a part of. Here *next* is a part of a node and it will point to the next node.

Now once we have created a definition for each node, we need to define the data type of *struct LinkedList* and create a new node as shown in the code snippet below.

```
1. typedef struct LinkedList * node; //Define node as pointer of data type struct LinkedList
2. node createNode() {
3.    node temp; // declare a node
4.    temp = (node) malloc(sizeof(struct LinkedList)); // allocate memory using malloc()
5.    temp - > next = NULL; // make next point to NULL
6.    return temp; //return the new node
7. }
```

Code Snippet 2.2: Creation of a node in Single Linked List

typedef is used to define a data type in C.

malloc() is used to dynamically allocate a single block of memory in C, it is available in the header file stdlib.h.

sizeof() is used to determine size in bytes of an element in C. Here it is used to determine size of each node and sent as a parameter to malloc.

At first we defined the node as pointer of data type. Then we define a function named *createNode* which can be used to create and initialize a node anytime in the program just by calling this function. In this function, we first declare a temporary node as *temp*. After declaration we need to allocate memory to these nodes manually. So for that initially we calculate the memory or the space that the

node requires by *sizeof(struct LinkedList)*. And then we allocate that amount of space or memory to that particular node using *malloc()* as follows:

temp = (node) malloc(sizeof(struct LinkedList));

Now that we have created a node as well as allocated memory to it, we need to assign the address of the next node in the *next* field of the node to create a link between them. But as of now we do not have any other nodes we assign NULL to it.

temp -> next = NULL

At the end we return this newly created node to wherever *createNode()* function was called. As it is a pointer type, it'll return the address of this newly create node. This address can be used to create a link with any previously existing node by storing the address in the *next* field of the previous node.

2.2.1.2 Insertion of a node:

A node can be inserted in three positions in a single linked list.

- Inserting node at the beginning of the list(head).
- Inserting node at any given position in the linked list.
- Inserting node at the end of the list(tail).

2.2.1.2.1 Inserting node at the Head:

The new node is added before the head of the given Linked List. And newly added node becomes the new head of the Linked List. For example, if the given Linked List is A->B->C->D and we add an item E at the front, then the Linked List becomes E->A->B->C->D. Let us call the function that adds at the front of the list is push(). The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node.

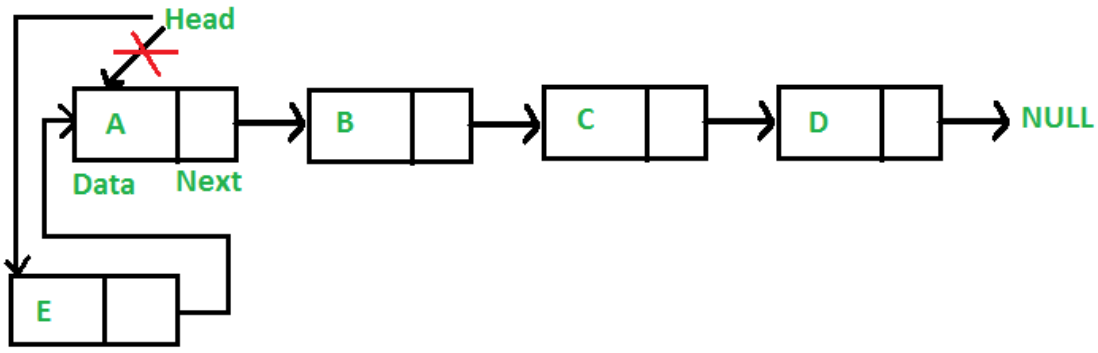


Figure 2.3: Inserting node at the head of Single Linked List

```
1.  /* Given a reference (pointer to pointer) to the head of a list and an int, inserts a new node on the fr
    ont of the list. */
2.  void push(struct Node * * head_ref, int new_data) { /* 1. allocate node */
3.      struct Node * new_node = (struct Node * ) malloc(sizeof(struct Node));      /* 2. put in the data  */
```

```
4.    new_node - > data  = new_data;      /* 3. Make next of new node as head */
5.    new_node - > next = ( * head_ref);      /* 4. move the head to point to the new node */
6.    ( * head_ref)    = new_node;
7.  }
```

Code Snippet 2.3: Inserting Node at the head of Single Linked List

Initially we create and allocate memory to a new node that we want to insert at the head of the linked list i.e., *new_node*. Then we store the data in it say 'E'. It is done by:

new_node -> data = new_data (Here new_data holds the data 'E')

Now we need to link it to the existing head node i.e., the node that has data 'A' which is done by:

*new_node -> next = (*head_ref)*

NOTE: Here head_ref contains the address of the head node i.e., the node with data 'A'

Now we need to assign the address of this newly created node to the *head_ref* so that it becomes the head which is done by:

*(*head_ref) = new_Node*

To insert a node at the head we just created a new node and store the address of the existing head node in the *next* field of the newly created node. Now change the address in *head_ref* to the address of this newly created node to make it as head.

2.2.1.2.2 Inserting node at a Given Position:

- Step 1: Initialize count c to 1
- Step 2: Create inserting node
  cur = (struct node*)malloc(sizeof (struct node));
- Step 3: Read the content of node
- Step 4: Read the position of insertion
- Step 5: Inserting in a given position
  next=first;
  repeat the steps a to c until c < pos
  prev=next;
  next=prev->link;
  c++;
  cur->link=prev->link;
  prev->link=cur;
- Step 6: Stop

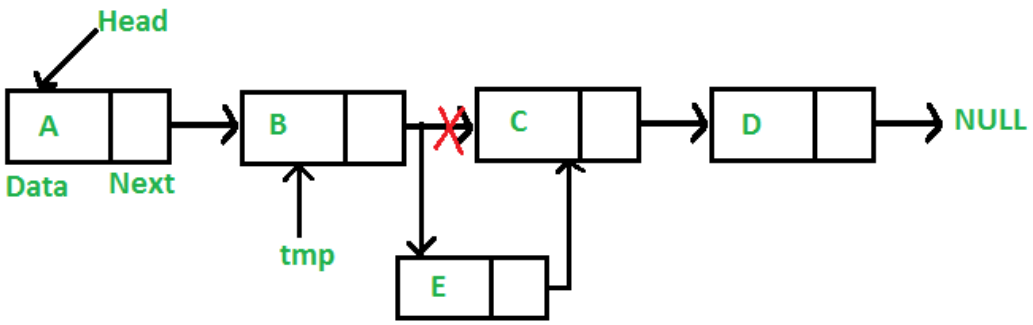


Fig 2.4: Inserting node at a given position in a Single Linked List

Initially we need to create and allocate memory to a new node that we want to insert. We need to store the data in it as well. Then we need to read the position from the user at which we need to insert this new node say the position given by user is 2. So, we need to insert the new node after 2nd node and before 3rd node i.e., between the nodes B and C as shown in Fig. 2.4.

So therefore, first we need to traverse to 2nd node, i.e., B, which contains the address to 3rd node i.e., C. Now we need to store the value in the *next* field of the 2nd node (which is the address of the 3rd node) in a temporary variable.

Now we need to change the value in the *next* field of the 2nd node with address of the new node that we want to insert. And after that we need to store the value i.e., the address of the 3rd node which we stored previously in a temporary variable, in the *next* field of the new node.

2.2.1.2.3 Inserting node at the Rear:

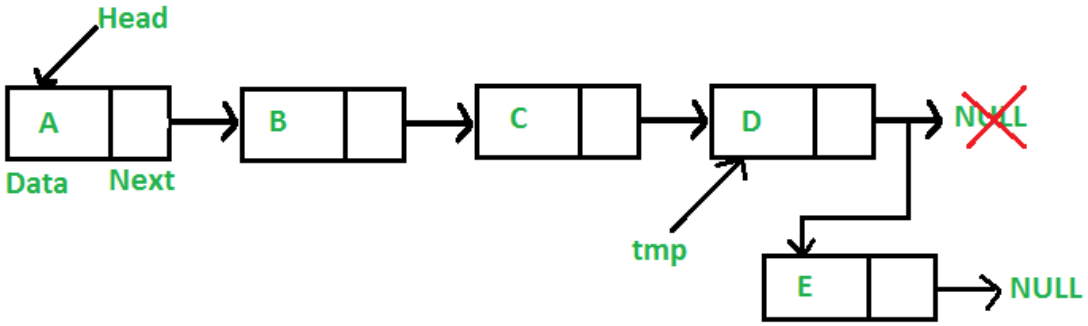


Fig 2.5: Inserting Node at the Rear

The new node is always added after the last node of the given Linked List. For example if the given Linked List A->B->C->D and we add an item E at the end, then the Linked List becomes A->B->C->D->E.

Since a Linked List is typically represented by the head of it, we have to traverse the list till end and then change the next of last node to new node.

```
1.  node addNode(node head, int value) {
2.      node temp, p; // declare two nodes temp and p
```

```
3.    temp = createNode(); //createNode will return a new node with data = value and next pointing to
NULL.
4.    temp - > data = value; // add element's value to data part of node
5.    if (head == NULL) {
6.       head = temp; //when linked list is empty
7.    } else {
8.       p = head; //assign head to p
9.       while (p - > next != NULL) {
10.         p = p - > next; //traverse the list until p is the last node.The last node always points to NULL.
11.      }
12.      p - > next = temp; //Point the previous last node to the new node created.
13.   }
14.   return head;
15. }
```

Code Snippet 2.4: Inserting node at the Rear

Initially we create and allocate memory to the new node which we need to insert the rear, say *temp.* Now we store the data in it and the traverse to the end of the list.

The end is known when p -> next == NULL

Now we assign replace NULL with the address of the node we need to replace. And then place NULL in the *next* field of the new node that is inserted at the end.

2.2.1.3 Deletion:

Deleting any node can be performed easily by traversing through the list to find the key i.e., the node to be deleted and then free the memory where that node is stored and link up the previous and next nodes.

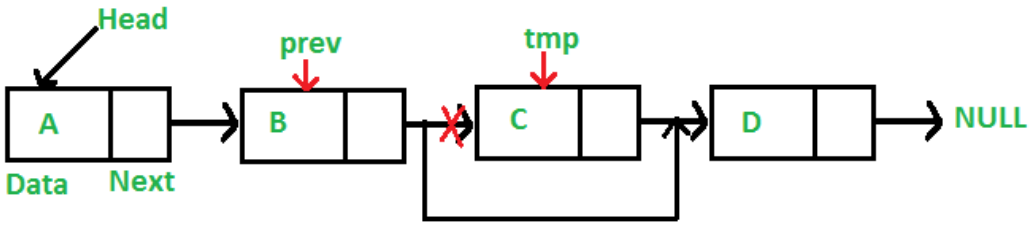


Fig 2.6: Deleting a node at a given key

Given, the key of the node *key*, to be deleted.

To delete a node from linked list, we need to do following steps.
1) Find previous node of the node to be deleted.
2) Changed next of previous node.
3) Free memory for the node to be deleted.

Since every node of linked list is dynamically allocated using malloc() in C, we need to call free() for freeing memory allocated for the node to be deleted.

```
1.  /* Given a reference (pointer to pointer) to the head of a list   and a key, deletes the first occurrence
    of key in linked list */
2.  void deleteNode(struct Node * * head_ref, int key) {
3.   // Store head node
4.     struct Node * temp = * head_ref, * prev;
5.
6.  // If head node itself holds the key to be deleted
7.     if (temp != NULL && temp - > data == key)
8.             {
9.             * head_ref = temp - > next;   // Changed head
10.       free(temp);             // free old head
11.       return;
12.    }
13.
14.   // Search for the key to be deleted, keep track of the previous node as we need to change 'prev-
    >next'
15.    while (temp != NULL && temp - > data != key)
16.            {
17.       prev = temp;
18.       temp = temp - > next;
19.    }
20.
21. // If key was not present in linked list
22.    if (temp == NULL) return;
23.
24. // Unlink the node from linked list
25.    prev - > next = temp - > next;
26.    free(temp);  // Free memory
27. }
```

Code Snippet 2.5: Deleting node in a single linked list

Initially we pass the key of the node that we need to delete along with the head node so that we can traverse through the list. While traversing through the list we search for the node that is to be deleted. Once we encounter the node that needs to be deleted, we delete it by freeing up the memory space it is stored in by using *free()*. Beforehand we store the address of the previous as well as the next node. We do that by storing the information in *temp.*

2.2.1.4 Traversal:

Traversing linked list means visiting each and every node of the Singly linked list. Following steps are involved while traversing the singly linked list –

1. Firstly, move to the first node
2. Fetch the data from the node and perform the operations such as arithmetic operation or any operation depending on data type.
3. After performing operation, advance pointer to next node and perform all above steps on Visited node.

```
struct node *temp= start;
do {
// Do Your Operation
// Statement ...1
// Statement ...2
// Statement ...3
// Statement ...n
   temp = temp - > next; //Move Pointer to Next Node
} while (temp != NULL);
```

Code Snippet 2.6: Traversing in a single linked list

Single Linked List Implementation & Operations

```
# include < stdio.h >
#include < conio.h >
#include < stdlib.h >

void insertAtBeginning(int);
void insertAtEnd(int);
void insertBetween(int, int, int);
void display();
void removeBeginning();
void removeEnd();
void removeSpecific(int);
struct Node {
   int data;
   struct Node * next;
} * head = NULL;
void main() {
   int choice, value, choice1, loc1, loc2;
   clrscr();
   while (1) {
      mainMenu: printf("\n\n********* MENU ************\n1. Insert\n2. Display\n3. Delete\n4. Exit\nEnter your choice: ");scanf("%d", & choice);
      switch (choice) {
         case 1:
            printf("Enter the value to be insert: ");
            scanf("%d", & value);
            while (1) {
               printf("Where you want to insert: \n1. At Beginning\n2. At End\n3. Between\nEnter your choice: ");
               scanf("%d", & choice1);
               switch (choice1) {
                  case 1:
                     insertAtBeginning(value);
                     break;
                  case 2:
                     insertAtEnd(value);
                     break;
                  case 3:
```

```
                    printf("Enter the two values where you wanto insert: ");
                    scanf("%d%d", & loc1, & loc2);
                    insertBetween(value, loc1, loc2);
                    break;
                default:
                    printf("\nWrong Input!! Try again!!!\n\n");
                    goto mainMenu;
            }
            goto subMenuEnd;
        }
        subMenuEnd: break;
      case 2:
        display();
        break;
      case 3:
        printf("How do you want to Delete: \n1. From Beginning\n2. From End\n3. Spesific\nEnter your choice: ");
        scanf("%d", & choice1);
        switch (choice1) {
          case 1:
            removeBeginning();
            break;
          case 2:
            removeEnd(value);
            break;
          case 3:
            printf("Enter the value which you wanto delete: ");
            scanf("%d", & loc2);
            removeSpecific(loc2);
            break;
          default:
            printf("\nWrong Input!! Try again!!!\n\n");
            goto mainMenu;
        }
        break;
      case 4:
        exit(0);
      default:
        printf("\nWrong input!!! Try again!!\n\n");
    }
  }
}
void insertAtBeginning(int value) {
  struct Node * newNode;
  newNode = (struct Node * ) malloc(sizeof(struct Node));
  newNode - > data = value;
  if (head == NULL) {
    newNode - > next = NULL;
    head = newNode;
  } else {
    newNode - > next = head;
```

```
86.       head = newNode;
87.    }
88.    printf("\nOne node inserted!!!\n");
89. }
90. void insertAtEnd(int value) {
91.    struct Node * newNode;
92.    newNode = (struct Node * ) malloc(sizeof(struct Node));
93.    newNode - > data = value;
94.    newNode - > next = NULL;
95.    if (head == NULL) head = newNode;
96.    else {
97.       struct Node * temp = head;
98.       while (temp - > next != NULL) temp = temp - > next;
99.       temp - > next = newNode;
100.           }
101.           printf("\nOne node inserted!!!\n");
102.        }
103.        void insertBetween(int value, int loc1, int loc2) {
104.           struct Node * newNode;
105.           newNode = (struct Node * ) malloc(sizeof(struct Node));
106.           newNode - > data = value;
107.           if (head == NULL) {
108.              newNode - > next = NULL;
109.              head = newNode;
110.           } else {
111.              struct Node * temp = head;
112.              while (temp - > data != loc1 && temp - > data != loc2) temp = temp - > next;
113.              newNode - > next = temp - > next;
114.              temp - > next = newNode;
115.           }
116.           printf("\nOne node inserted!!!\n");
117.        }
118.        void removeBeginning() {
119.           if (head == NULL) printf("\n\nList is Empty!!!");
120.           else {
121.              struct Node * temp = head;
122.              if (head - > next == NULL) {
123.                 head = NULL;
124.                 free(temp);
125.              } else {
126.                 head = temp - > next;
127.                 free(temp);
128.                 printf("\nOne node deleted!!!\n\n");
129.              }
130.           }
131.        }
132.        void removeEnd() {
133.           if (head == NULL) {
134.              printf("\nList is Empty!!!\n");
135.           } else {
136.              struct Node * temp1 = head, * temp2;
```

```
137.            if (head - > next == NULL) head = NULL;
138.            else {
139.               while (temp1 - > next != NULL) {
140.                  temp2 = temp1;
141.                  temp1 = temp1 - > next;
142.               }
143.               temp2 - > next = NULL;
144.            }
145.            free(temp1);
146.            printf("\nOne node deleted!!!\n\n");
147.         }
148.      }
149.      void removeSpecific(int delValue) {
150.         struct Node * temp1 = head, * temp2;
151.         while (temp1 - > data != delValue) {
152.            if (temp1 - > next == NULL) {
153.               printf("\nGiven node not found in the list!!!");
154.               goto functionEnd;
155.            }
156.            temp2 = temp1;
157.            temp1 = temp1 - > next;
158.         }
159.         temp2 - > next = temp1 - > next;
160.         free(temp1);
161.         printf("\nOne node deleted!!!\n\n");
162.         functionEnd:
163.      }
164.      void display() {
165.         if (head == NULL) {
166.            printf("\nList is Empty\n");
167.         } else {
168.            struct Node * temp = head;
169.            printf("\n\nList elements are - \n");
170.            while (temp - > next != NULL) {
171.               printf("%d --->", temp - > data);
172.               temp = temp - > next;
173.            }
174.            printf("%d --->NULL", temp - > data);
175.         }
176.      }
```

2.3 Double Linked Lists

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 link of a linked list can store a data called an element.
- Next − Each link of a linked list contains a link to the next link called Next.
- Prev − Each link of a linked list contains a link to the previous link called Prev.

- LinkedList − A Linked List contains the connection link to the first link called First and to the last link called Last.

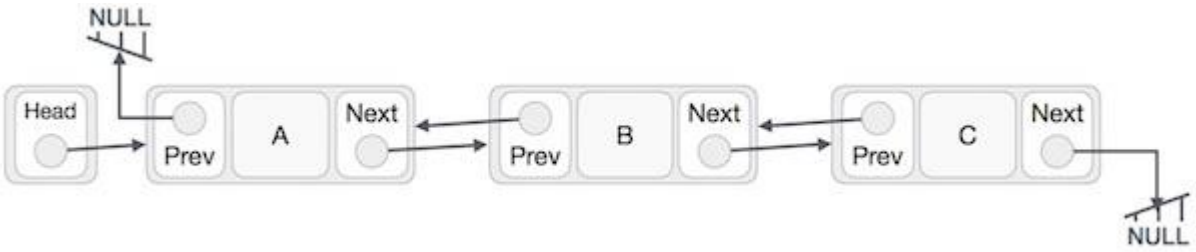


Fig 2.7: Double Linked List Representation

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 link carries a data field(s) and two link fields called next and prev.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.

In double linked list, every node has link to its previous node and next node. So, we can traverse forward by using next field and can traverse backward by using previous field. Every node in a double linked list contains three fields *prev*, *data* and *next.* The fields *prev* and *next* are known as *links.*

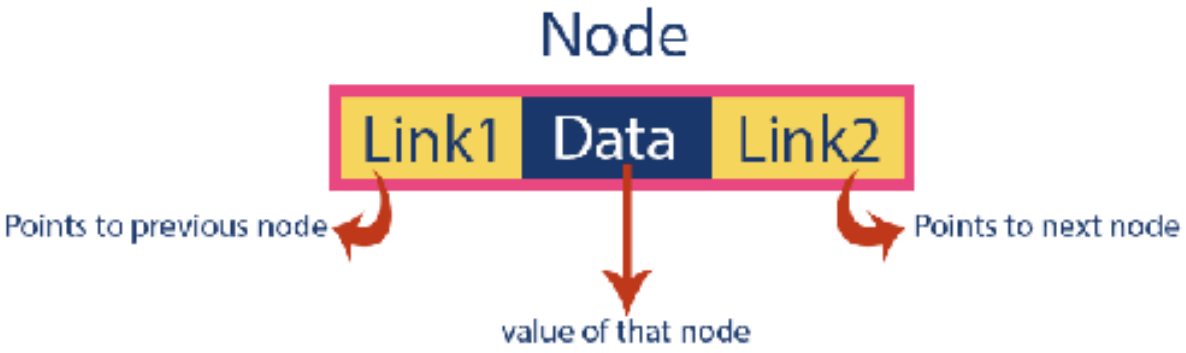


Fig 2.8: Double Linked List Node

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.

NOTE:

- **In double linked list, the first node must be always pointed by head.**
- **Always the previous field of the first node must be NULL.**
- **Always the next field of the last node must be NULL.**

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

- Insertion
- Deletion
- Traversal

2.3.1 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.

2.3.1.1 Insertion at the beginning:

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.

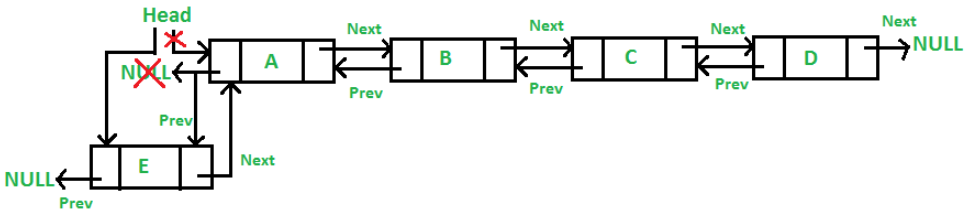


Fig 2.9: Inserting node at the beginning of the Double Linked List

```
/* Given a reference (pointer to pointer) to the head of a list   and an int, inserts a new node on the f
ront of the list. */
void push(struct Node * * head_ref, int new_data)
        {
        /* 1. allocate node */
  struct Node * new_node = (struct Node * ) malloc(sizeof(struct Node));

        /* 2. put in the data  */
  new_node - > data  = new_data;

/* 3. Make next of new node as head and previous as NULL */
  new_node - > next = ( * head_ref);
  new_node - > prev = NULL;

/* 4. change prev of head node to new node */
  if (( * head_ref) !=  NULL)
   ( * head_ref) - > prev = new_node;

```

```
/* 5. move the head to point to the new node */
   ( * head_ref)    = new_node;
}
```

Code Snippet 2.7: Inserting node at the beginning of DLL

First, we create and allocate memory to a new double linked list node and store data in it. Now as we need to insert it at the beginning, we can simply store the address of the head node in the *next* field of this new node and store NULL in the *prev* field of this new node as this will be the new head. Now we need to store the address of this new node in the *prev* field of the existing head node. Then we need to assign address of the new node to *head_ref* to mark it as the new head.

2.3.1.2 Insertion at the end:

The new node is added after the last node of the given Linked List. For example, if the given DLL is ABCD and we add an item E at the end, then the DLL becomes ABCDE. Since a Linked List is typically represented by the head of it, we have to traverse the list till end and then change the next of last node to new node.

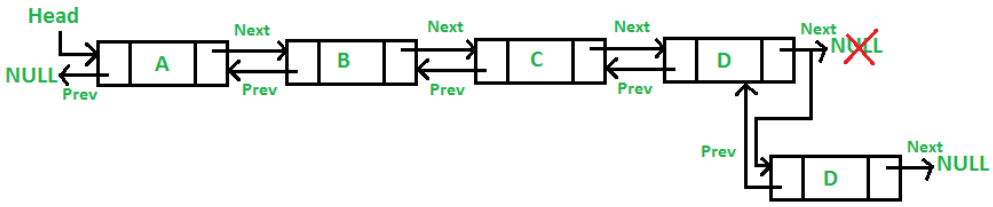


Fig 2.10: Inserting node at the End of the Double Linked List

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.

```
/* Given a reference (pointer to pointer) to the headof a DLL and an int, appends a new node at the end */
void append(struct Node * * head_ref, int new_data) { /* 1. allocate node */
   struct Node * new_node = (struct Node * ) malloc(sizeof(struct Node));
   struct Node * last = * head_ref; /* used in step 5*/ /* 2. put in the data */
   new_node - > data = new_data;
   /* 3. This new node is going to be the last node, so      make next of it as NULL*/
   new_node - > next = NULL;
   /* 4. If the Linked List is empty, then make the new      node as head */
   if ( * head_ref == NULL) {
      new_node - > prev = NULL; * head_ref = new_node;
      return;
```

```
12.    } /* 5. Else traverse till the last node */
13.    while (last - > next != NULL) last = last - > next; /* 6. Change the next of last node */
14.    last - > next = new_node; /* 7. Make last node as previous of new node */
15.    new_node - > prev = last;
16.    return;
17. }
```

Code Snippet 2.8: Inserting node at the end of DLL

2.3.1.2 Insertion at the given node:

We are given pointer to a node as prev_node, and the new node is inserted after the given node.

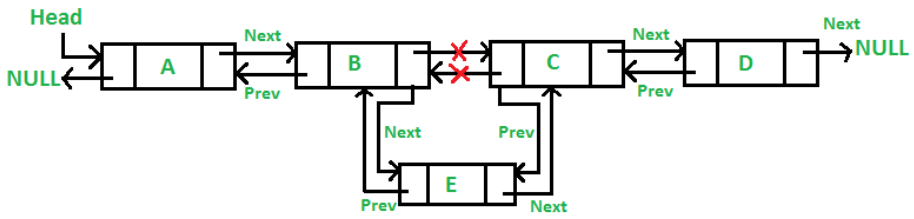


Fig 2.11: Inserting node at a given node of the Double Linked List

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 newNode → previous & newNode → next and 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**.

```
1.  void insertAtAnyPosition(int x,int pos)
2.  {
3.     struct node* temp=NULL;
4.     temp=(node*)malloc(sizeof(struct node));
5.     temp->data=x;
6.     if(head==NULL)
7.     {
8.        temp->next=NULL;
9.        head=temp;
10.       temp->prev=head;
```

```
11.    }
12.    else
13.    {
14.        struct node* temp1=head;
15.        struct node* temp2=NULL;
16.       int i=0;
17.       for(i=0;i<pos-1;i++)
18.       {
19.          temp1=temp1->next;
20.          Temp2=temp1->next;
21.       }
22.       temp->next=temp2;
23.       Temp1->prev=temp->next;
24.       Temp1->next=temp;
25.       temp->prev=temp1;
26.    }
27. }
```

Code Snippet 2.9: Inserting node at a given node of DLL

2.3.2 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

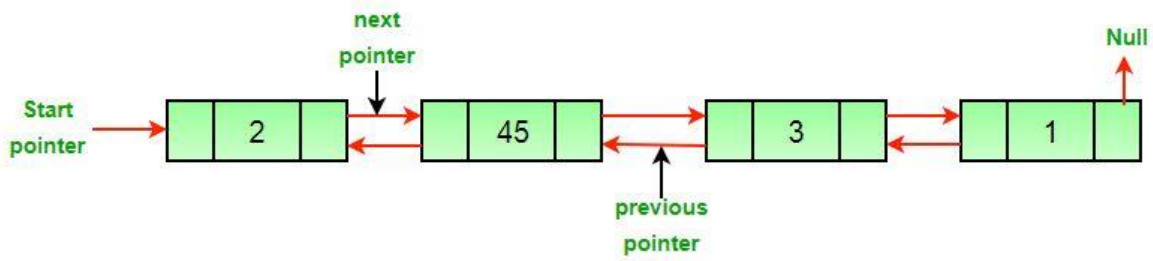


Fig 2.12: Original Double Linked List

Now let's say we need to delete the second node i.e., node which contains 45 as data in the figure 2.12. For deleting we assign the address of the 3rd node next pointer to the 1st node and address of the 1st node to the previous pointer of the 3rd node. And then free up the memory allocated to the 2nd node to delete it.

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.

```
1.  /* Function to delete a node in a Doubly Linked List.   head_ref --
    > pointer to head node pointer.   del  -->  pointer to node to be deleted. */
2.  void deleteNode(struct Node * * head_ref, struct Node * del)
3.  {    /* base case */
4.     if ( * head_ref == NULL || del == NULL)
5.          return;
6.  /* If node to be deleted is head node */
7.     if ( * head_ref == del)
8.          * head_ref = del - > next;
9.
10.      /* Change next only if node to be deleted is NOT the last node */
11.    if (del - > next != NULL)
12.                        del - > next - > prev = del - > prev;
13.         /* Change prev only if node to be deleted is NOT the first node */
14.    if (del - > prev != NULL)
15.         del - > prev - > next = del - > next;
16.
17.     /* Finally, free the memory occupied by del*/
18.    free(del);
19. }
```

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)).

```
/* Function to delete the node at the given position   in the doubly linked list */
void deleteNodeAtGivenPos(struct Node * * head_ref, int n)
 {
   /* if list in NULL or invalid position is given */
    if ( * head_ref == NULL || n <= 0)
        return;
    struct Node * current = * head_ref;
    int i;
    /* traverse up to the node at position 'n' from      the beginning */

    for (int i = 1; current != NULL && i < n; i++)
        current = current - > next;
    /* if 'n' is greater than the number of nodes      in the doubly linked list */
    if (current == NULL)
        return;
     /* delete the node pointed to by 'current' */
    deleteNode(head_ref, current);
}
```

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.

```
void deleteFromEnd() {
    struct node * toDelete;
    if (last == NULL) {
        printf("Unable to delete. List is empty.\n");
    } else {
```

```
        toDelete = last;
        last = last - > prev; // Move last pointer to 2nd last node
        last - > next = NULL; // Remove link to of 2nd last node with last node
        free(toDelete); // Delete the last node
        printf("SUCCESSFULLY DELETED NODE FROM END OF THE LIST.\n");
    }
}
```

2.3.2 Traversing:

Traversing linked list means visiting each and every node of the double linked list. We can use the following steps to display the elements of a double linked list by traversing:

- 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).

```
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);
    }
```

Double Linked List Implementation & Operations

```
#include < stdio.h >
#include < conio.h >

void insertAtBeginning(int);
void insertAtEnd(int);
void insertAtAfter(int, int);
void deleteBeginning();
void deleteEnd();
void deleteSpecific(int);
void display();
struct Node {
    int data;
    struct Node * previous, * next;
} * head = NULL;
void main() {
    int choice1, choice2, value, location;
```

```
17.  clrscr();
18.  while (1) {
19.    printf("\n********** MENU ************\n");
20.    printf("1. Insert\n2. Delete\n3. Display\n4. Exit\nEnter your choice: ");
21.    scanf("%d", & choice1);
22.    switch () {
23.      case 1:
24.        printf("Enter the value to be inserted: ");
25.        scanf("%d", & value);
26.        while (1) {
27.          printf("\nSelect from the following Inserting options\n");
28.          printf("1. At Beginning\n2. At End\n3. After a Node\n4. Cancel\nEnter your choice: ");
29.          scanf("%d", & choice2);
30.          switch (choice2) {
31.            case 1:
32.              insertAtBeginning(value);
33.              break;
34.            case 2:
35.              insertAtEnd(value);
36.              break;
37.            case 3:
38.              printf("Enter the location after which you want to insert: ");
39.              scanf("%d", & location);
40.              insertAfter(value, location);
41.              break;
42.            case 4:
43.              goto EndSwitch;
44.            default:
45.              printf("\nPlease select correct Inserting option!!!\n");
46.          }
47.        }
48.      case 2:
49.        while (1) {
50.          printf("\nSelect from the following Deleting options\n");
51.          printf("1. At Beginning\n2. At End\n3. Specific Node\n4. Cancel\nEnter your choice: ");
52.          scanf("%d", & choice2);
53.          switch (choice2) {
54.            case 1:
55.              deleteBeginning();
56.              break;
57.            case 2:
58.              deleteEnd();
59.              break;
60.            case 3:
61.              printf("Enter the Node value to be deleted: ");
62.              scanf("%d", & location);
63.              deleteSpecic(location);
64.              break;
65.            case 4:
66.              goto EndSwitch;
67.            default:
```

```
68.                     printf("\nPlease select correct Deleting option!!!\n");
69.                 }
70.             }
71.             EndSwitch: break;
72.         case 3:
73.             display();
74.             break;
75.         case 4:
76.             exit(0);
77.         default:
78.             printf("\nPlease select correct option!!!");
79.       }
80.    }
81. }
82. void insertAtBeginning(int value) {
83.    struct Node * newNode;
84.    newNode = (struct Node * ) malloc(sizeof(struct Node));
85.    newNode - > data = value;
86.    newNode - > previous = NULL;
87.    if (head == NULL) {
88.       newNode - > next = NULL;
89.       head = newNode;
90.    } else {
91.       newNode - > next = head;
92.       head = newNode;
93.    }
94.    printf("\nInsertion success!!!");
95. }
96. void insertAtEnd(int value) {
97.    struct Node * newNode;
98.    newNode = (struct Node * ) malloc(sizeof(struct Node));
99.    newNode - > data = value;
100.            newNode - > next = NULL;
101.            if (head == NULL) {
102.               newNode - > previous = NULL;
103.               head = newNode;
104.            } else {
105.               struct Node * temp = head;
106.               while (temp - > next != NULL) temp = temp - > next;
107.               temp - > next = newNode;
108.               newNode - > previous = temp;
109.            }
110.            printf("\nInsertion success!!!");
111.         }
112.         void insertAfter(int value, int location) {
113.            struct Node * newNode;
114.            newNode = (struct Node * ) malloc(sizeof(struct Node));
115.            newNode - > data = value;
116.            if (head == NULL) {
117.               newNode - > previous = newNode - > next = NULL;
118.               head = newNode;
```

```
119.          } else {
120.             struct Node * temp1 = head, temp2;
121.             while (temp1 - > data != location) {
122.                if (temp1 - > next == NULL) {
123.                   printf("Given node is not found in the list!!!");
124.                   goto EndFunction;
125.                } else {
126.                   temp1 = temp1 - > next;
127.                }
128.             }
129.             temp2 = temp1 - > next;
130.             temp1 - > next = newNode;
131.             newNode - > previous = temp1;
132.             newNode - > next = temp2;
133.             temp2 - > previous = newNode;
134.             printf("\nInsertion success!!!");
135.          }
136.          EndFunction:
137.       }
138.       void deleteBeginning() {
139.          if (head == NULL) printf("List is Empty!!! Deletion not possible!!!");
140.          else {
141.             struct Node * temp = head;
142.             if (temp - > previous == temp - > next) {
143.                head = NULL;
144.                free(temp);
145.             } else {
146.                head = temp - > next;
147.                head - > previous = NULL;
148.                free(temp);
149.             }
150.             printf("\nDeletion success!!!");
151.          }
152.       }
153.       void deleteEnd() {
154.          if (head == NULL) printf("List is Empty!!! Deletion not possible!!!");
155.          else {
156.             struct Node * temp = head;
157.             if (temp - > previous == temp - > next) {
158.                head = NULL;
159.                free(temp);
160.             } else {
161.                while (temp - > next != NULL) temp = temp - > next;
162.                temp - > previous - > next = NULL;
163.                free(temp);
164.             }
165.             printf("\nDeletion success!!!");
166.          }
167.       }
168.       void deleteSpecific(int delValue) {
169.          if (head == NULL) printf("List is Empty!!! Deletion not possible!!!");
```

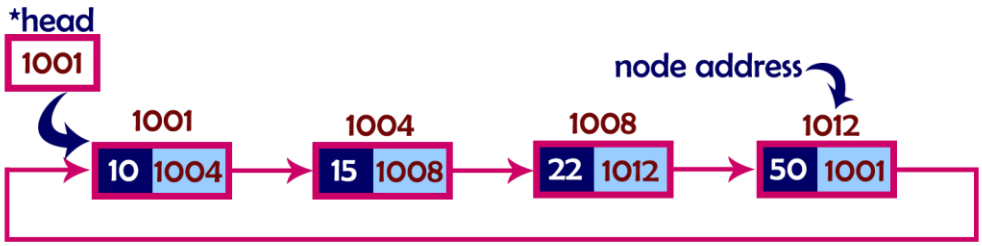
```
170.         else {
171.            struct Node * temp = head;
172.            while (temp - > data != delValue) {
173.               if (temp - > next == NULL) {
174.                  printf("\nGiven node is not found in the list!!!");
175.                  goto FuctionEnd;
176.               } else {
177.                  temp = temp - > next;
178.               }
179.            }
180.            if (temp == head) {
181.               head = NULL;
182.               free(temp);
183.            } else {
184.               temp - > previous - > next = temp - > next;
185.               free(temp);
186.            }
187.            printf("\nDeletion success!!!");
188.         }
189.         FuctionEnd:
190.      }
191.      void display() {
192.         if (head == NULL) printf("\nList is Empty!!!");
193.         else {
194.            struct Node * temp = head;
195.            printf("\nList elements are: \n");
196.            printf("NULL <--- ");
197.            while (temp - > next != NULL) {
198.               printf("%d <===> ", temp - > data);
199.            }
200.            printf("%d ---> NULL", temp - > data);
201.         }
202.      }
```

2.4 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.

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

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




```
1. struct Node {
2.    int data;
3.    struct Node * next;
4. } * head = NULL;
```

Initially we need to create and define node so that it could be used later.

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

1. Insertion
2. Deletion
3. Display

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'.

```
1. void insertAtBeginning(int value) {
2.    struct Node * newNode;
3.    newNode = (struct Node * ) malloc(sizeof(struct Node));
4.    newNode - > data = value;
5.    if (head == NULL) {
6.       head = newNode;
7.       newNode - > next = head;
8.    } else {
9.       struct Node * temp = head;
10.      while (temp - > next != head) temp = temp - > next;
```

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

```
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!!!");
                goto EndFunction;
            } else {
                temp = temp - > next;
            }
        }
        newNode - > next = temp - > next;
        temp - > next = newNode;
    }
}
```

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.

```
1.  void insertAtEnd(int value) {
2.     struct Node * newNode;
3.     newNode = (struct Node * ) malloc(sizeof(struct Node));
4.     newNode - > data = value;
5.     if (head == NULL) {
6.        head = newNode;
7.        newNode - > next = head;
8.     } else {
9.        struct Node * temp = head;
10.       while (temp - > next != head) temp = temp - > next;
11.       temp - > next = newNode;
12.       newNode - > next = head;
13.    }
14. }
```

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.

```
1.  void deleteBeginning() {
2.     if (head == NULL) printf("List is Empty!!! Deletion not possible!!!");
3.     else {
4.        struct Node * temp = head;
5.        if (temp - > next == head) {
6.           head = NULL;
7.           free(temp);
8.        } else {
9.           head = head - > next;
10.          free(temp);
11.       }
12.    }
13. }
```

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 → next = 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 11: 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)).

```
1.  void deleteSpecific(int delValue) {
2.     if (head == NULL) printf("List is Empty!!! Deletion not possible!!!");
3.     else {
4.        struct Node * temp1 = head, temp2;
5.        while (temp1 - > data != delValue) {
6.           if (temp1 - > next == head) {
7.              printf("\nGiven node is not found in the list!!!");
8.              goto FuctionEnd;
```

```
        } 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);
        }
      }
    }
}
```

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.

```
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 {
```

```
9.          while (temp1 - > next != head) {
10.            temp2 = temp1;
11.            temp1 = temp1 - > next;
12.         }
13.         temp2 - > next = head;
14.         free(temp1);
15.      }
16.   }
17. }
```

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.

```
1.  void display() {
2.     if (head == NULL) printf("\nList is Empty!!!");
3.     else {
4.        struct Node * temp = head;
5.        printf("\nList elements are: \n");
6.        while (temp - > next != head) {
7.           printf("%d ---> ", temp - > data);
8.        }
9.        printf("%d ---> %d", temp - > data, head - > data);
10.    }
11. }
```

Circular Linked List Implementation & Operations

```
1.  #include < stdio.h >
2.  #include < conio.h >
3.
4.  void insertAtBeginning(int);
5.  void insertAtEnd(int);
6.  void insertAtAfter(int, int);
7.  void deleteBeginning();
8.  void deleteEnd();
9.  void deleteSpecific(int);
10. void display();
11. struct Node {
12.    int data;
13.    struct Node * next;
14. } * head = NULL;
15. void main() {
16.    int choice1, choice2, value, location;
17.    clrscr();
```

```
18.   while (1) {
19.     printf("\n********** MENU *************\n");
20.     printf("1. Insert\n2. Delete\n3. Display\n4. Exit\nEnter your choice: ");
21.     scanf("%d", & choice1);
22.     switch () {
23.       case 1:
24.         printf("Enter the value to be inserted: ");
25.         scanf("%d", & value);
26.         while (1) {
27.           printf("\nSelect from the following Inserting options\n");
28.           printf("1. At Beginning\n2. At End\n3. After a Node\n4. Cancel\nEnter your choice: ");
29.           scanf("%d", & choice2);
30.           switch (choice2) {
31.             case 1:
32.               insertAtBeginning(value);
33.               break;
34.             case 2:
35.               insertAtEnd(value);
36.               break;
37.             case 3:
38.               printf("Enter the location after which you want to insert: ");
39.               scanf("%d", & location);
40.               insertAfter(value, location);
41.               break;
42.             case 4:
43.               goto EndSwitch;
44.             default:
45.               printf("\nPlease select correct Inserting option!!!\n");
46.           }
47.         }
48.       case 2:
49.         while (1) {
50.           printf("\nSelect from the following Deleting options\n");
51.           printf("1. At Beginning\n2. At End\n3. Specific Node\n4. Cancel\nEnter your choice: ");
52.           scanf("%d", & choice2);
53.           switch (choice2) {
54.             case 1:
55.               deleteBeginning();
56.               break;
57.             case 2:
58.               deleteEnd();
59.               break;
60.             case 3:
61.               printf("Enter the Node value to be deleted: ");
62.               scanf("%d", & location);
63.               deleteSpecic(location);
64.               break;
65.             case 4:
66.               goto EndSwitch;
67.             default:
68.               printf("\nPlease select correct Deleting option!!!\n");
```

```
                }
            }
            EndSwitch: break;
        case 3:
            display();
            break;
        case 4:
            exit(0);
        default:
            printf("\nPlease select correct option!!!");
    }
  }
}
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!!!");
}
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!!!");
        }
        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;
```

```
120.        } else {
121.           struct Node * temp = head;
122.           while (temp - > data != location) {
123.              if (temp - > next == head) {
124.                 printf("Given node is not found in the list!!!");
125.                 goto EndFunction;
126.              } else {
127.                 temp = temp - > next;
128.              }
129.           }
130.           newNode - > next = temp - > next;
131.           temp - > next = newNode;
132.           printf("\nInsertion success!!!");
133.        }
134.        EndFunction:
135.     }
136.     void deleteBeginning() {
137.        if (head == NULL) printf("List is Empty!!! Deletion not possible!!!");
138.        else {
139.           struct Node * temp = head;
140.           if (temp - > next == head) {
141.              head = NULL;
142.              free(temp);
143.           } else {
144.              head = head - > next;
145.              free(temp);
146.           }
147.           printf("\nDeletion success!!!");
148.        }
149.     }
150.     void deleteEnd() {
151.        if (head == NULL) printf("List is Empty!!! Deletion not possible!!!");
152.        else {
153.           struct Node * temp1 = head, temp2;
154.           if (temp1 - > next == head) {
155.              head = NULL;
156.              free(temp1);
157.           } else {
158.              while (temp1 - > next != head) {
159.                 temp2 = temp1;
160.                 temp1 = temp1 - > next;
161.              }
162.              temp2 - > next = head;
163.              free(temp1);
164.           }
165.           printf("\nDeletion success!!!");
166.        }
167.     }
168.     void deleteSpecific(int delValue) {
169.        if (head == NULL) printf("List is Empty!!! Deletion not possible!!!");
170.        else {
```

```
171.            struct Node * temp1 = head, temp2;
172.            while (temp1 - > data != delValue) {
173.               if (temp1 - > next == head) {
174.                  printf("\nGiven node is not found in the list!!!");
175.                  goto FuctionEnd;
176.               } else {
177.                  temp2 = temp1;
178.                  temp1 = temp1 - > next;
179.               }
180.            }
181.            if (temp1 - > next == head) {
182.               head = NULL;
183.               free(temp1);
184.            } else {
185.               if (temp1 == head) {
186.                  temp2 = head;
187.                  while (temp2 - > next != head) temp2 = temp2 - > next;
188.                  head = head - > next;
189.                  temp2 - > next = head;
190.                  free(temp1);
191.               } else {
192.                  if (temp1 - > next == head) {
193.                     temp2 - > next = head;
194.                  } else {
195.                     temp2 - > next = temp1 - > next;
196.                  }
197.                  free(temp1);
198.               }
199.            }
200.            printf("\nDeletion success!!!");
201.         }
202.         FuctionEnd:
203.      }
204.      void display() {
205.         if (head == NULL) printf("\nList is Empty!!!");
206.         else {
207.            struct Node * temp = head;
208.            printf("\nList elements are: \n");
209.            while (temp - > next != head) {
210.               printf("%d ---> ", temp - > data);
211.            }
212.            printf("%d ---> %d", temp - > data, head - > data);
213.         }
214.      }
```