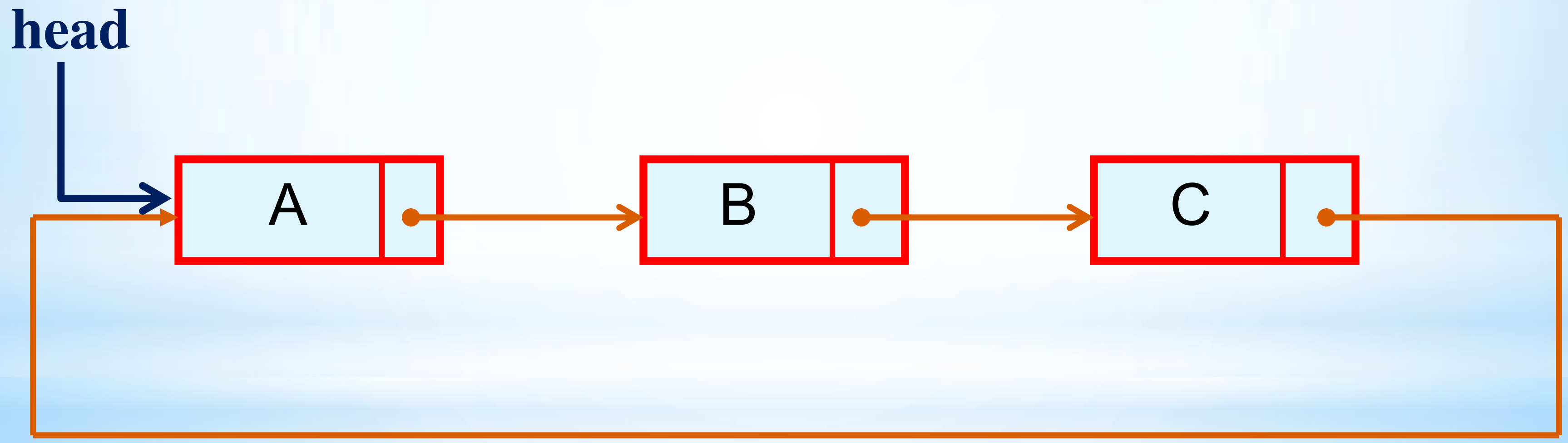
Types of Lists: Circular and Double Linked List

Types of Lists: Circular Linked List

Circular linked list

- The pointer from the last element in the list points back to the first element.



Circular Linked List

- A circular linked list is basically a linear linked list that may be single- or double-linked.
- The only difference is that there is no any NULL value terminating the list.
- In fact in the list every node points to the next node and last node points to the first node, thus forming a circle. Since it forms a circle with no end to stop it is called as **circular linked list**.
- In circular linked list there can be no starting or ending node, whole node can be traversed from any node.
- In order to traverse the circular linked list, only once we need to traverse entire list until the starting node is not traversed again.
- A circular linked list can be implemented using both singly linked list and doubly linked list.

Circular linked list:

Advantages of a Circular linked list

- Entire list can be traversed from any node.
- Circular lists are the required data structure when we want a list to be accessed in a circle or loop.
- Despite of being singly circular linked list we can easily traverse to its previous node, which is not possible in singly linked list.

Disadvantages of Circular linked list

- Circular list are complex as compared to singly linked lists.
- Reversing of circular list is a complex as compared to singly or doubly lists.
- If not traversed carefully, then we could end up in an infinite loop.
- Like singly and doubly lists circular linked lists also doesn’t supports direct accessing of elements.

Operations on circular linked list

- Creation of list
- Traversal of list
- Insertion of node
    - At the beginning of list
    - At any position in the list
- Deletion of node
    - Deletion of first node
    - Deletion of node from middle of the list
    - Deletion of last node
- Counting total number of nodes
- Reversing of list

Creation and Traversal of a Circular List

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

/* Basic structure of Node */

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

int main()
{
    int n, data;
    head = NULL;

    printf("Enter the total number of nodes in list: ");
    scanf("%d", &n);
    createList(n);                    // function to create circular linked list
    displayList();                    // function to display the list

    return 0;
}
```

Circular Linked List: Creation of List

```
void createList(int n)
{
    int i, data;
    struct node *prevNode, *newNode;
    if(n >= 1){                      /* Creates and links the head node */
        head = (struct node *)malloc(sizeof(struct node));

        printf("Enter data of 1 node: ");
        scanf("%d", &data);

        head->data = data;
        head->next = NULL;
        prevNode = head;

        for(i=2; i<=n; i++){       /* Creates and links rest of the n-1 nodes */
            newNode = (struct node *)malloc(sizeof(struct node));

            printf("Enter data of %d node: ", i);
            scanf("%d", &data);

            newNode->data = data;
            newNode->next = NULL;
            prevNode->next = newNode; //Links the previous node with newly created node
            prevNode = newNode;      //Moves the previous node ahead
        }
            prevNode->next = head; //Links the last node with first node
           printf("\nCIRCULAR LINKED LIST CREATED SUCCESSFULLY\n");
    }
}
```

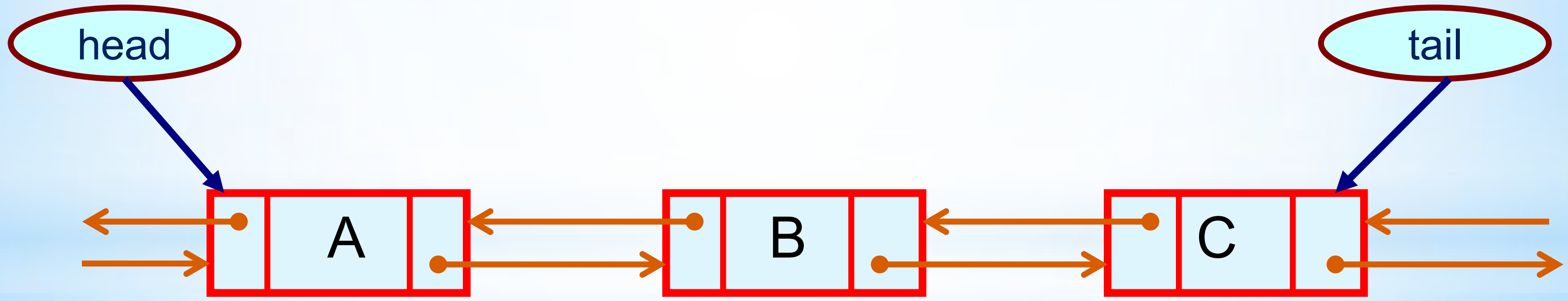
Circular Linked List: Traversal of List

```
void displayList()
{
    struct node *current;
    int n = 1;

    if(head == NULL)
    {
        printf("List is empty.\n");
    }
    else
    {
        current = head;
        printf("DATA IN THE LIST:\n");

        do {
            printf("Data %d = %d\n", n, current->data);

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

Types of Lists: Double Linked List

Double linked list

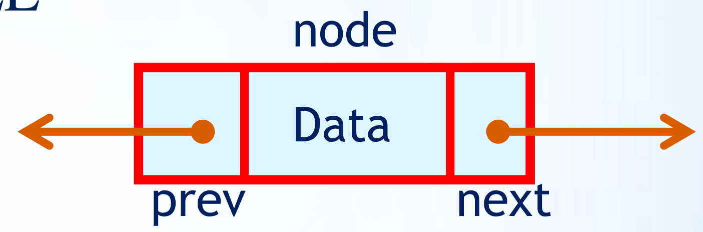
- Pointers exist between adjacent nodes in both directions.
- The list can be traversed either forward or backward.
- Usually two pointers are maintained to keep track of the list, *head* and *tail*.



Defining a Node of a Double Linked List

Each node of doubly linked list (DLL) consists of three fields:

- Item (or) Data
- Pointer of the next node in DLL
- Pointer of the previous node in DLL

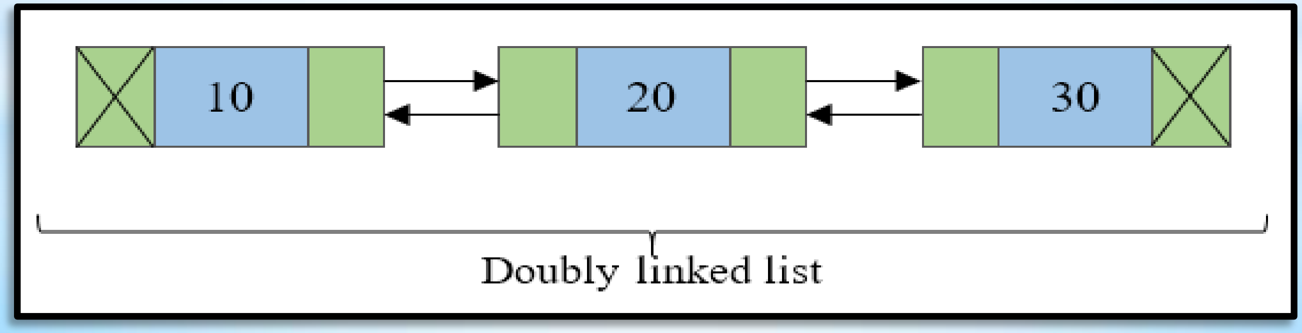


How to define a node of a doubly linked list (DLL)?

```
struct node
{
  int data;
  struct node *next; // Pointer to next node in DLL
  struct node *prev; // Pointer to previous node in DLL
};
```

Double Linked List

- Doubly linked list is a collection of nodes linked together in a sequential way.
- Doubly linked list is almost similar to singly linked list except it contains two address or reference fields, where one of the address field contains reference of the next node and other contains reference of the previous node.
- First and last node of a linked list contains a terminator generally a NULL value, that determines the start and end of the list.
- Doubly linked list is sometimes also referred as bi-directional linked list since it allows traversal of nodes in both direction.
- Since doubly linked list allows the traversal of nodes in both direction, we can keep track of both first and last nodes.



Double versus Single Linked List

Advantages over singly linked list

1) A DLL can be traversed in both forward and backward direction.
2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.

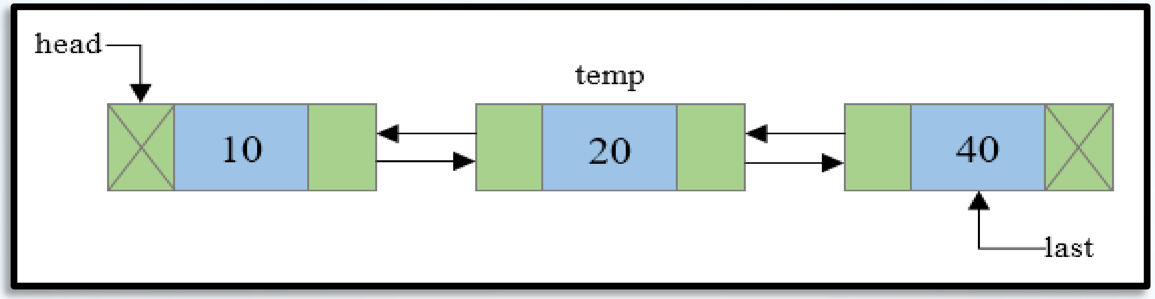
Disadvantages over singly linked list

1) Every node of DLL Require extra space for an previous pointer.
2) All operations require an extra pointer previous to be maintained.

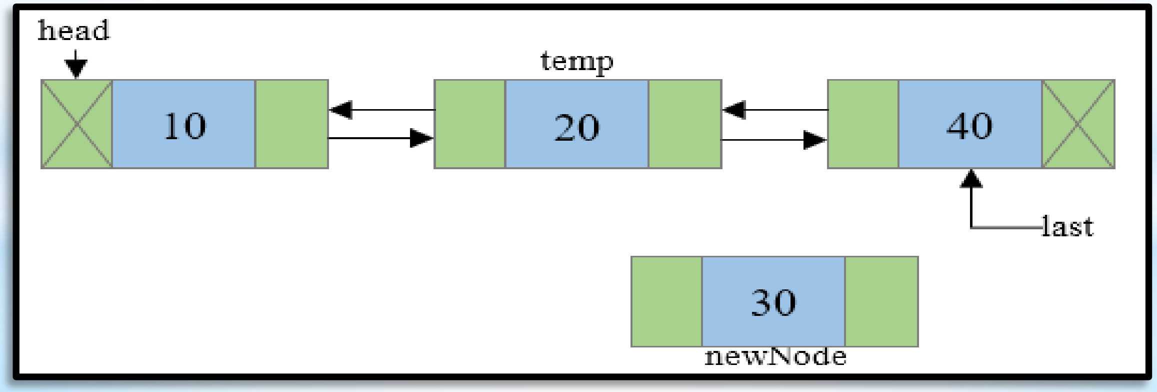
Double Linked List: Insertion at any Position

Steps to insert a new node at nth position in a Doubly linked list.

Step 1: Traverse to N-1 node in the list, where N is the position to insert. Say **`temp`** now points to N-1th node.

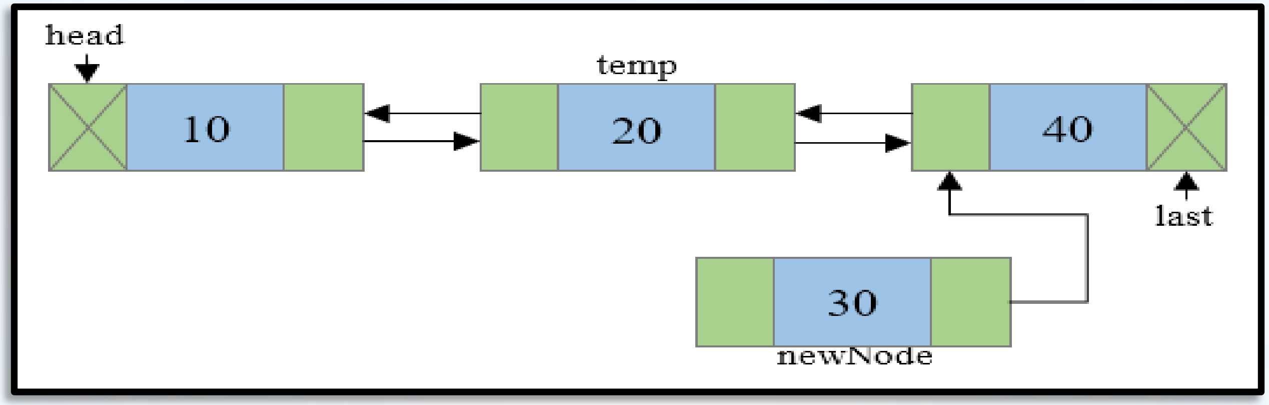


Step 2: Create a **`newNode`** that is to be inserted and assign some data to its data field.

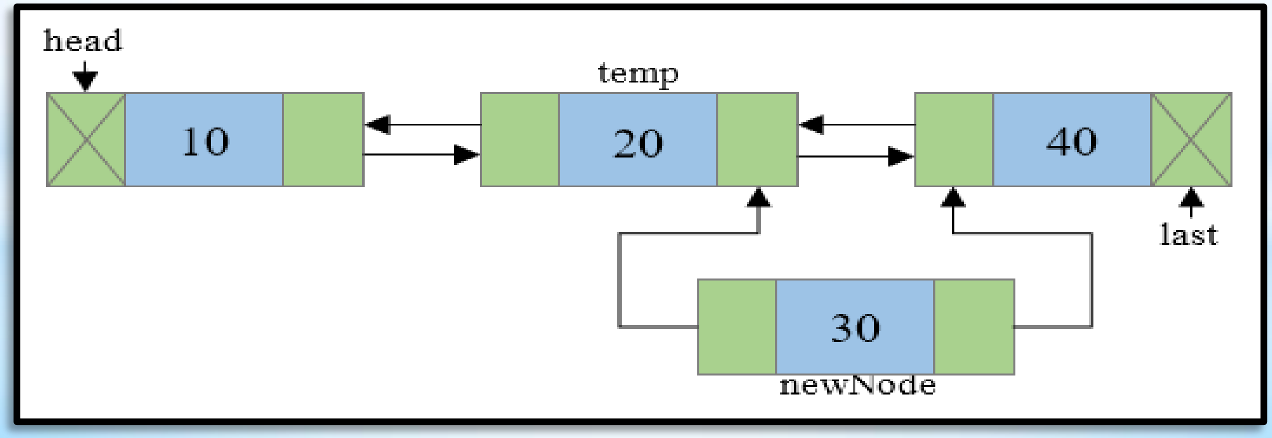


Doubly Linked List: Insertion at any Position

Step 3: Connect the next address field of **`newNode`** with the node pointed by next address field of **`temp`** node.

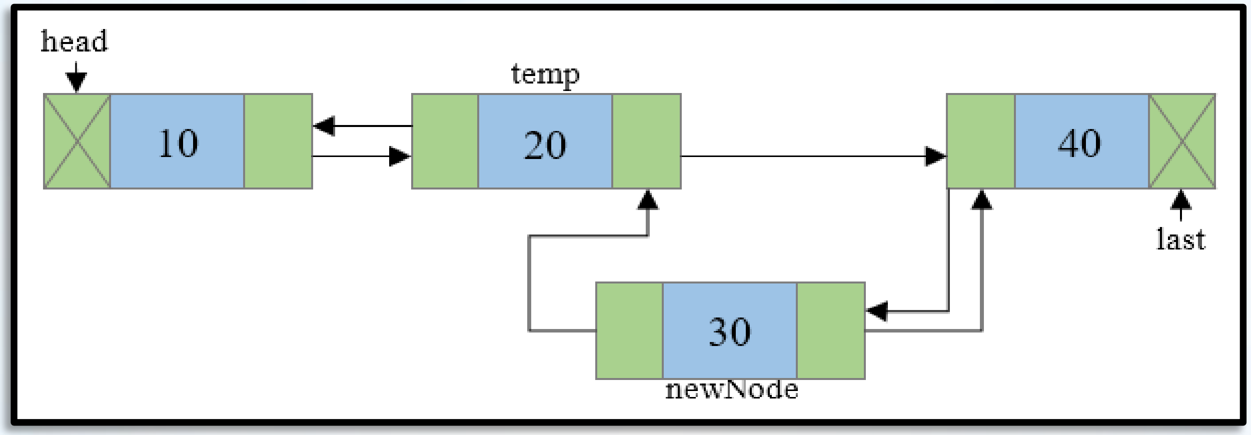


Step 4: Connect the previous address field of **`newNode`** with the **`temp`** node.

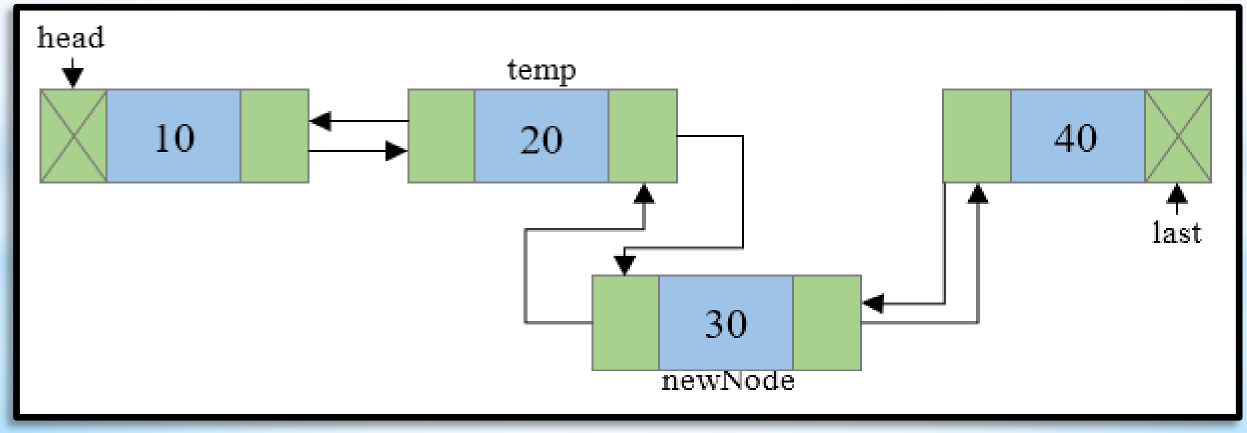


Doubly Linked List: Insertion at any Position

Step 5: Check if `temp.next` is not `NULL` then, connect the previous address field of node pointed by `temp.next` to `newNode`.

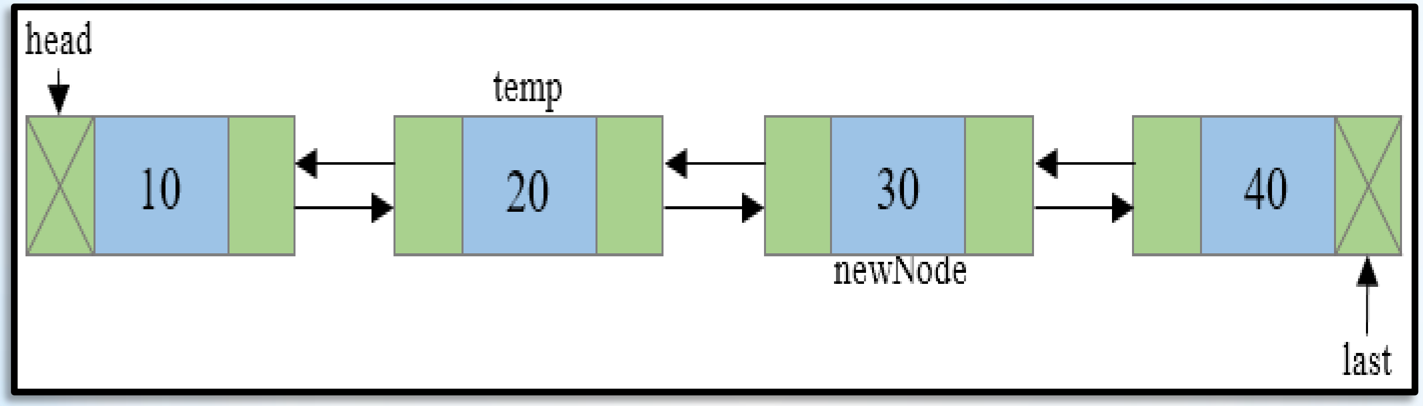


Step 6: Connect the next address field of `temp` node to `newNode`.



Doubly Linked List: Insertion at any Position

Step 7: Final doubly linked list looks like



Doubly Linked List: Insertion at any Position

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

struct node {                   /* Basic structure of Node */
    int data;
    struct node * prev;
    struct node * next;
}*head, *last;

int main()
{
    int n, data;
    head = NULL;
    last = NULL;

    printf("Enter the total number of nodes in list: ");
    scanf("%d", &n);
    createList(n);                        // function to create double linked list
    displayList();                        // function to display the list

    printf("Enter the position and data to insert new node: ");
    scanf("%d %d", &n, &data);
    insert_position(data, n);     // function to insert node at any position
    displayList();
    return 0;
}
```

Doubly Linked List: Insertion at any Position

```
void createList(int n)
{
    int i, data;
    struct node *newNode;
    if(n >= 1){                 /* Creates and links the head node */
        head = (struct node *)malloc(sizeof(struct node));
        printf("Enter data of 1 node: ");
        scanf("%d", &data);
        head->data = data;
        head->prev = NULL;
        head->next = NULL;

        last = head;

        for(i=2; i<=n; i++){     /* Creates and links rest of the n-1 nodes */
            newNode = (struct node *)malloc(sizeof(struct node));
            printf("Enter data of %d node: ", i);
            scanf("%d", &data);

            newNode->data = data;
            newNode->prev = last;       //Links new node with the previous node
            newNode->next = NULL;

            last->next = newNode; //Links previous node with the new node
            last = newNode; //Makes new node as last/previous node
        }
        printf("\nDOUBLY LINKED LIST CREATED SUCCESSFULLY\n");
    }
}
```

Doubly Linked List: Insertion at any Position

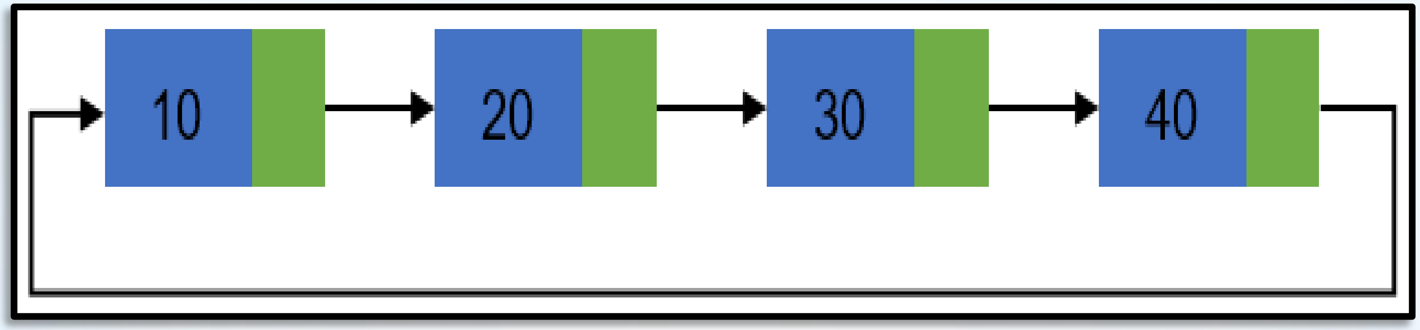
```
void insert_position(int data, int position)
{
    struct node * newNode, *temp; int count=1;
    if(head == NULL){
        printf("Error, List is empty!\n");
    }
    else{
        temp = head;
        if(temp!=NULL){
            newNode = (struct node *)malloc(sizeof(struct node));
            While(count<position) temp=temp->next;
            newNode->data = data;
            newNode->next = temp->next; //Connects new node with n+1th node
            newNode->prev = temp;       //Connects new node with n-1th node

            if(temp->next != NULL)
            {
                temp->next->prev = newNode; /* Connects n+1th node with new node */
            }
            temp->next = newNode;          /* Connects n-1th node with new node */
            printf("NODE INSERTED SUCCESSFULLY AT %d POSITION\n", position);
        }
        else{
            printf("Error, Invalid position\n");
        }
    }
}
```

Doubly Linked List: Insertion at any Position

```
void displayList()
{
    struct node * temp;
    int n = 1;

    if(head == NULL)
    {
        printf("List is empty.\n");
    }
    else
    {
        temp = head;
        printf("DATA IN THE LIST:\n");

        while(temp != NULL)
        {
            printf("DATA of %d node = %d\n", n, temp->data);
            n++;

            /* Moves the current pointer to next node */
            temp = temp->next;
        }
    }
}
```

Circular linked list:

Basic structure of singly circular linked list:



Doubly circular linked list: