

### Unit III:

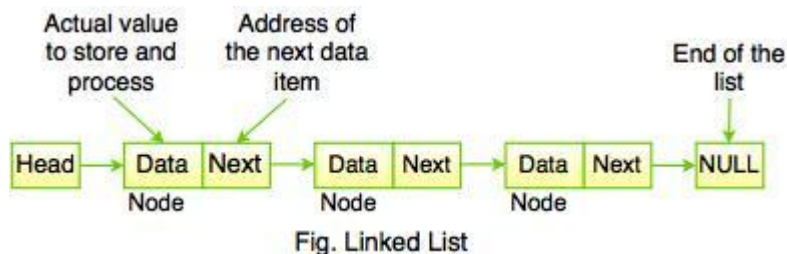
**Lists:** Lists – Linked List – Singly linked list – doubly linked list – Circular linked list – Representation of Stacks using linked lists – Representation of Queues using linked lists– Applications of Linked list.

## What is Linked List?

Linked list is a linear data structure. It is a collection of data elements, called nodes pointing to the next node by means of a pointer.

Linked list is used to create trees and graphs.

In linked list, each node consists of its own data and the address of the next node and forms a chain.



The above figure shows the sequence of linked list which contains data items connected together via links. It can be visualized as a chain of nodes, where every node points to the next node.

Linked list contains a link element called **first** and each link carries a **data item**. Entry point into the linked list is called the **head of the list**.

Link field is called next and each link is linked with its next link. Last link carries a link to null to mark the end of the list.

**Note:** Head is not a separate node but it is a reference to the first node. If the list is empty, the head is a null reference.

Linked list is a dynamic data structure. While accessing a particular item, start at the head and follow the references until you get that data item.

**Linked list is used while dealing with an unknown number of objects:**



In the above diagram, Linked list contains two fields - First field contains value and second field contains a link to the next node. The last node signifies the end of the list that means NULL.

The real life **example of Linked List** is that of Railway Carriage. It starts from engine and then the coaches follow. Coaches can traverse from one coach to other, if they connected to each other.

### Advantages of Linked List

- Linked list is dynamic in nature which allocates the memory when required.
- In linked list, stack and queue can be easily executed.
- It reduces the access time.
- Insert and delete operation can be easily implemented in linked list.

### Disadvantages of Linked List

- Reverse traversing is difficult in linked list.
- Linked list has to access each node sequentially; no element can be accessed randomly.
- In linked list, the memory is wasted as pointer requires extra memory for storage.

Example: Program to create a simple linked list.

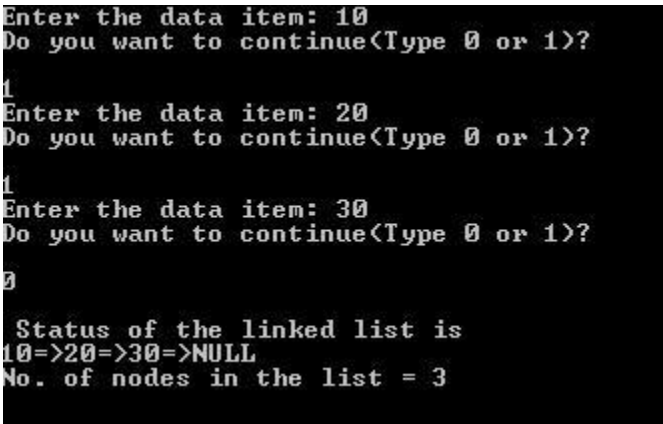
```
#include<stdio.h>
#include <stdlib.h>
int main()
{
    struct node
    {
        int num;
        struct node *ptr;
    };
    typedef struct node NODE;
    NODE *head, *first, *temp=0;
    int count = 0;
    int choice = 1;
    first = 0;
    while(choice)
    {
        head =(NODE*) malloc(sizeof(NODE));
        printf("Enter the data item: ");
        scanf("%d", &head-> num);
        if(first != 0)
        {
            temp->ptr = head;temp = head;
        }
        else
        {
            first = temp = head;
        }
        fflush(stdin);
```

```

    printf("Do you want to continue(Type 0 or 1)?\n\n");
    scanf("%d", &choice);
}
temp->ptr = 0;
temp = first; /* reset temp to the beginning*/
printf("\n Status of the linked list is\n");
while(temp!=0)
{
    printf("%d=>", temp->num);
    count++;
    temp = temp -> ptr;
}
printf("NULL\n");
printf("No. of nodes in the list = %d\n", count);
return 0;
}

```

### Output:



```

Enter the data item: 10
Do you want to continue(Type 0 or 1)?
1
Enter the data item: 20
Do you want to continue(Type 0 or 1)?
1
Enter the data item: 30
Do you want to continue(Type 0 or 1)?
0

Status of the linked list is
10=>20=>30=>NULL
No. of nodes in the list = 3

```

## Types of Linked List

Following are the types of Linked List

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List
4. Doubly Circular Linked List

# 1. Singly Linked List

- Each node has a single link to another node is called Singly Linked List.
- Singly Linked List does not store any pointer any reference to the previous node.
- Each node stores the contents of the node and a reference to the next node in the list.
- In a singly linked list, last node has a pointer which indicates that it is the last node. It requires a reference to the first node to store a single linked list.
- It has two successive nodes linked together in linear way and contains address of the next node to be followed.
- It has successor and predecessor. First node does not have predecessor while last node does not have successor. Last node have successor reference as NULL.
- It has only single link for the next node.
- In this type of linked list, only forward sequential movement is possible, no direct access is allowed.

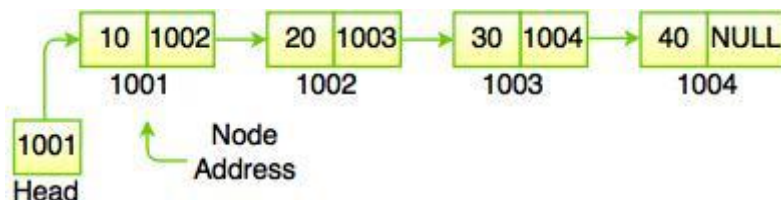


Fig. Singly Linked List

- In the above figure, the address of the first node is always store in a reference node known as Head or Front. Reference part of the last node must be null.

Example: Program to perform various operations on Singly Linked List

```
#include<stdio.h>
#include<stdlib.h>
typedef struct Node
{
    int data;
    struct Node *next;
}node;
void insert(node *pointer, int data)
{
    /* Iterate through the list till we encounter the last node.*/
    while(pointer->next!=NULL)
    {
        pointer = pointer -> next;
    }
    /* Allocate memory for the new node and put data in it.*/
```

```

pointer->next = (node *)malloc(sizeof(node));
pointer = pointer->next;
pointer->data = data;
pointer->next = NULL;
}
int find(node *pointer, int key)
{
    pointer = pointer -> next; //First node is dummy node.
    /* Iterate through the entire linked list and search for the key. */
    while(pointer!=NULL)
    {
        if(pointer->data == key) //key is found.
        {
            return 1;
        }
        pointer = pointer -> next;//Search in the next node.
    }
    /*Key is not found */
    return 0;
}
void delete(node *pointer, int data)
{
    /* Go to the node for which the node next to it has to be deleted */
    while(pointer->next!=NULL && (pointer->next)->data != data)
    {
        pointer = pointer -> next;
    }
    if(pointer->next==NULL)
    {
        printf("Element %d is not present in the list\n",data);
        return;
    }
    /* Now pointer points to a node and the node next to it has to be removed */
    node *temp;
    temp = pointer -> next;
    /*temp points to the node which has to be removed*/
    pointer->next = temp->next;
    /*We removed the node which is next to the pointer (which is also temp) */
    free(temp);
}

```

```

    /* Because we deleted the node, we no longer require the memory used for it .
    free() will deallocate the memory.
    */
    return;
}
void print(node *pointer)
{
    if(pointer==NULL)
    {
        return;
    }
    printf("%d ",pointer->data);
    print(pointer->next);
}
int main()
{
    /* start always points to the first node of the linked list.
    temp is used to point to the last node of the linked list.*/
    node *start,*temp;
    start = (node *)malloc(sizeof(node));
    temp = start;
    temp -> next = NULL;
    /* Here in this code, we take the first node as a dummy node.
    The first node does not contain data, but it used because to avoid handling special cases in
    insert and delete functions. */
    printf("1. Insert\n");
    printf("2. Delete\n");
    printf("3. Print\n");
    printf("4. Find\n");
    printf("Enter your choice: ");
    while(1)
    {
        int query;
        scanf("%d",&query);
        if(query==1)
        {
            int data;
            printf("Enter the element: ");
            scanf("%d",&data);

```

```

        insert(start,data);
        printf("Press 3 for display: ");
    }
    else if(query==2)
    {
        int data;
        printf("Delete the element: ");
        scanf("%d",&data);
        delete(start,data);
        printf("Press 3 for display: ");
    }
    else if(query==3)
    {
        printf("The list is ");
        print(start->next);
        printf("\n");
    }
    else if(query==4)
    {
        int data;
        printf("Find the element: ");
        scanf("%d",&data);
        int status = find(start,data);
        if(status)
        {
            printf("Element Found\n");
        }
        else
        {
            printf("Element Not Found\n");
        }
    }
}
}
}

```

**Output:**

```

1. Insert
2. Delete
3. Print
4. Find
Enter your choice: 1
Enter the element: 10
Press 3 for display: 3
The list is 10
1
Enter the element: 20
Press 3 for display: 3
The list is 10 20
1
Enter the element: 30
Press 3 for display: 3
The list is 10 20 30
2
Delete the element: 20
Press 3 for display: 3
The list is 10 30
4
Find the element: 10
Element Found

```

## 2. Doubly Linked List

- Doubly linked list is a sequence of elements in which every node has link to its previous node and next node.
- Traversing can be done in both directions and displays the contents in the whole list.



Fig. Doubly Linked List

In the above figure, Link1 field stores the address of the previous node and Link2 field stores the address of the next node. The Data Item field stores the actual value of that node. If we insert a data into the linked list, it will be look like as follows:

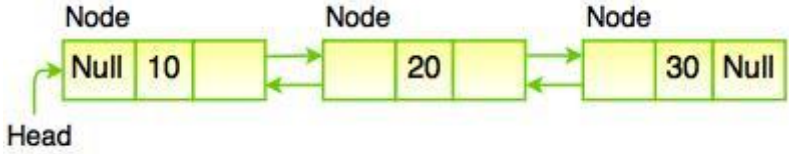


Fig. Doubly Linked List

**Important Note:**

First node is always pointed by head. In doubly linked list, previous field of the first node is always NULL (it must be NULL) and the next field of the last must be NULL.

In the above figure we see that, doubly linked list contains three fields. In this, link of two nodes allow traversal of the list in either direction. There is no need to traverse the list to find the previous node. We can traverse from head to tail as well as tail to head.



### **Advantages of Doubly Linked List**

- Doubly linked list can be traversed in both forward and backward directions.
- To delete a node in singly linked list, the previous node is required, while in doubly linked list, we can get the previous node using previous pointer.
- It is very convenient than singly linked list. Doubly linked list maintains the links for bidirectional traversing.

### **Disadvantages of Doubly Linked List**

- In doubly linked list, each node requires extra space for previous pointer.
- All operations such as Insert, Delete, Traverse etc. require extra previous pointer to be maintained.

Example: Program to perform various operations on Doubly Linked List

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

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

```
typedef struct Node
```

```
{
```

```
    int data;
```

```
    struct Node *next;
```

```
    struct Node *prev;
```

```
}node;
```

```
void insert(node *pointer, int data)
```

```
{
```

```
    /* Iterate through the list till we encounter the last node.*/
```

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

```
    {
```

```
        pointer = pointer -> next;
```

```
    }
```

```
    /* Allocate memory for the new node and put data in it.*/
```

```
    pointer->next = (node *)malloc(sizeof(node));
```

```
    (pointer->next)->prev = pointer;
```

```
    pointer = pointer->next;
```

```
    pointer->data = data;
```

```
    pointer->next = NULL;
```

```
}
```

```
int find(node *pointer, int key)
```

```
{
```

```

pointer = pointer -> next; //First node is dummy node.
/* Iterate through the entire linked list and search for the key. */
while(pointer!=NULL)
{
    if(pointer->data == key) //key is found.
    {
        return 1;
    }
    pointer = pointer -> next; //Search in the next node.
}
/*Key is not found */
return 0;
}

void delete(node *pointer, int data)
{
    /* Go to the node for which the node next to it has to be deleted */
    while(pointer->next!=NULL && (pointer->next)->data != data)
    {
        pointer = pointer -> next;
    }
    if(pointer->next==NULL)
    {
        printf("Element %d is not present in the list\n",data);
        return;
    }
    /* Now pointer points to a node and the node next to it has to be removed */

    node *temp;
    temp = pointer -> next;

    /*temp points to the node which has to be removed*/
    pointer->next = temp->next;
    temp->prev = pointer;
    /*We removed the node which is next to the pointer (which is also temp) */
    free(temp);
    /* Because we deleted the node, we no longer require the memory used for it .
    free() will deallocate the memory. */
    return;
}

```

```

}
void print(node *pointer)
{
    if(pointer==NULL)
    {
        return;
    }
    printf("%d ",pointer->data);
    print(pointer->next);
}

```

```

int main()

```

```

{
    /* start always points to the first node of the linked list.
    temp is used to point to the last node of the linked list.*/
    node *start,*temp;
    start = (node *)malloc(sizeof(node));
    temp = start;
    temp -> next = NULL;
    temp -> prev = NULL;
    /* Here in this code, we take the first node as a dummy node.
    The first node does not contain data, but it used because to avoid handling special cases
    in insert and delete functions.
    */
    printf("1. Insert\n");
    printf("2. Delete\n");
    printf("3. Print\n");
    printf("4. Find\n");
    printf("Enter Choice: ");
    while(1)
    {
        int query;
        scanf("%d",&query);
        if(query==1)
        {
            int data;
            printf("Enter the element: ");
            scanf("%d",&data);
            insert(start,data);

```



```

1. Insert
2. Delete
3. Print
4. Find
Enter Choice: 1
Enter the element: 10
Enter 3 for display: 3
The list is: 10
1
Enter the element: 20
Enter 3 for display: 3
The list is: 10 20
1
Enter the element: 30
Enter 3 for display: 3
The list is: 10 20 30
2
Delete the element: 20
Enter 3 for display: 3
The list is: 10 30

```

### 3. Circular Linked List

- Circular linked list is similar to singly linked list. The only difference is that in circular linked list, the last node points to the first node in the list.
- It is a sequence of elements in which every element has link to its next element in the sequence and has a link to the first element in the sequence.

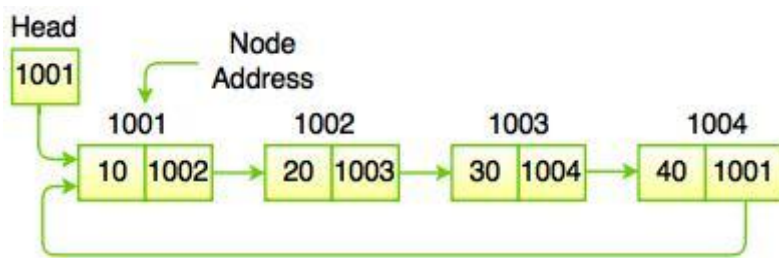


Fig. Circular Linked List

- In the above figure we see that, each node points to its next node in the sequence but the last node points to the first node in the list. The previous element stores the address of the next element and the last element stores the address of the starting element. It forms a circular chain because the element points to each other in a circular way.
- In circular linked list, the memory can be allocated when it is required because it has a dynamic size.
- Circular linked list is used in personal computers, where multiple applications are running. The operating system provides a fixed time slot for all running applications and the running applications are kept in a circular linked list until all the applications are completed. This is a real life example of circular linked list.
- We can insert elements anywhere in circular linked list, but in the array we cannot insert elements anywhere in the list because it is in the contiguous memory.

## 4. Doubly Circular Linked List

- Doubly circular linked list is a linked data structure which consists of a set of sequentially linked records called nodes.
- Doubly circular linked list can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders.

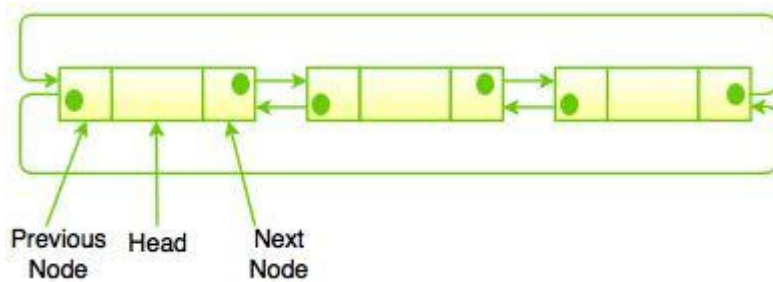


Fig. Doubly Circular Linked List

- The above diagram represents the basic structure of Doubly Circular Linked List. In doubly circular linked list, the previous link of the first node points to the last node and the next link of the last node points to the first node.
- In doubly circular linked list, each node contains two fields called links used to represent references to the previous and the next node in the sequence of nodes.

Example: Program to perform various operations on Doubly Circular Linked List

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

int data;
struct DCLL
{
    int info;
    struct DCLL *rptr, *lptr;
}*start = NULL, *temp, *neww, *prev, *end = NULL;

void ins_beg_dcll();
void ins_end_dcll();
void del_beg_dcll();
void del_end_dcll();
void create_dcll();
void travers();
```

```
void insert();  
void delet();  
void search();  
void update();
```

```
void main()  
{  
    int choice;  
    char ch, ch2 = 'y';  
    do  
    {  
        if (start == NULL)  
        {  
            do  
            {  
                // clrscr();  
                printf("\nError : Linked List is Empty Want to Create (Y/N).. ");  
                fflush(stdin);  
                scanf("%c", &ch);  
                if (ch == 'n' || ch == 'N')  
                {  
                    exit(0);  
                }  
                else if (ch == 'y' || ch == 'Y')  
                {  
                    create_dll();  
                    break;  
                }  
                else  
                {  
                    printf("\nError : Invalid Choice");  
                }  
            } while (ch == 'y' || ch == 'Y');  
        }  
        else  
        {  
            // clrscr();  
            printf("\n1. Insert");  
            printf("\n2. Delete");
```

```

printf("\n3. Traverse");
printf("\n4. Search");
printf("\n5. Update");
printf("\n\nPress 0 to Exit");
printf("\nEnter your Choice : ");
scanf("%d", &choice);

switch (choice)
{
    case 0:
        exit(0);
        break;
    case 1:
        insert();
        travers();
        break;
    case 2:
        delet();
        travers();
        break;
    case 3:
        travers();
        break;
    case 4:
        search();
        break;
    case 5:
        update();
        break;
    default:
        printf("\nError : Invalid Choice. \n");
}
printf("\nWant to Continue (Y/N)... ");
fflush(stdin);
scanf("%c", &ch2);
}

}while (ch2 == 'y' || ch2 == 'Y');
}

```



```

void create_dcll()
{
    char ch;
    do
    {
        //clrscr();
        printf("\nEnter Node Info");
        printf("\n\nEnter Info : ");
        scanf("%d", &data);
        temp = (struct DCLL *)malloc(sizeof(struct DCLL));
        temp->info = data;
        temp->rptr = start;
        temp->lptr = start;

        if (start == NULL)
        {
            start = temp;
            end = temp;
            start->rptr = start;
            start->lptr = start;
        }
        else
        {
            temp->lptr = end;
            end->rptr = temp;
            temp->rptr = start;
            start->lptr = temp;
            end = temp;
        }
        printf("\nDo you want to insert another Node (Y/N)... ");
        fflush(stdin);
        scanf("%c", &ch);
    } while (ch == 'y' || ch == 'Y');
}

void ins_beg_dcll()
{
    neww->rptr = start;
    start->lptr = neww;
    neww->lptr = end;
}

```

```

        end->rptr = neww;
        start = neww;
    }
void ins_end_dcll()
{
    neww->lptr = end;
    end->rptr = neww;
    neww->rptr = start;
    start->lptr = neww;
    end = neww;
}
void travers()
{
    temp = start;
    if (start == NULL)
    {
        printf("\nError : Linked list is Empty");
    }
    else
    {
        if (start == end)
        {
            printf("\n%d", temp->info);
        }
        else
        {
            printf("\n%d", temp->info);
            do
            {
                temp = temp->rptr;
                printf("\n%d", temp->info);
            } while (temp != end);
        }
    }
}
void insert()
{
    int pos, count = 1;
    printf("\nEnter Position : ");

```

```

scanf("%d", &pos);
printf("\nEnter Info :");
scanf("%d", &data);

neww = (struct DCLL *)malloc(sizeof(struct DCLL));
neww->info = data;
temp = start;

if (pos == 1)
{
    ins_beg_dcll();
}
else
{
    while (count != pos - 1 && temp->rptr != start)
    {
        count++;
        temp = temp->rptr;
    }
    if (temp->rptr == start && count < pos - 1)
    {
        printf("\nError : Invalid Position...");
    }
    else if (temp->rptr == start && count == pos - 1)
    {
        ins_end_dcll();
    }
    else
    {
        neww->rptr = temp->rptr;
        neww->lptr = temp;
        temp->rptr->lptr = neww;
        temp->rptr = neww;
    }
}
}

void delet()
{
    int pos, count = 1;

```

```

printf("\nEnter Position : ");
scanf("%d", &pos);
temp = start;
if (pos == 1)
{
    del_beg_dcll();
}
else
{
    while (count != pos && temp != end)
    {
        prev = temp;
        temp = temp->rptr;
        count++;
    }
    if (temp == end && count < pos - 1)
    {
        printf("\nError : Invalid Position.");
    }
    else if (temp == end && count == pos)
    {
        del_end_dcll();
    }
    else
    {
        prev->rptr = temp->rptr;
        temp->rptr->lptr = prev;
        free(temp);
    }
}
}
void del_beg_dcll()
{
    start = start->rptr;
    start->lptr = end;
    end->rptr = start;
    free(temp);
}
void del_end_dcll()

```

```

{
    end = end->lptr;
    end->rptr = start;
    start->lptr = end;
    free(temp);
}
void search()
{
    int src, count = 1;
    printf("\nEnter Info : ");
    scanf("%d", &src);
    temp = start;
    do
    {
        prev = temp;
        if (prev->info == src)
        {
            printf("\nSearch Value found at %d Position.", count);
            break;
        }
        count++;
        temp = temp->rptr;
    } while (prev != end);
    if (temp == start)
    {
        printf("\nError : Search value doesn't exists.");
    }
}
void update()
{
    int pos, count = 1, dt;
    printf("\nEnter Position : ");
    scanf("%d", &pos);
    printf("\nEnter New Info : ");
    scanf("%d", &dt);
    temp = prev = start;
    do
    {
        if (count == pos && prev != end)

```

```

    {
        prev->info = dt;
        break;
    }
    prev = temp;
    temp = temp->rptr;
    count++;
} while (prev != end);

if (temp == start)
{
    printf("\nError : Invalid Choice ");
}
}

```

## Output:

### 1. Create

```

Error : Linked List is Empty Want to Create <Y/N>.. y
Enter Node Info
Enter Info : 10
Do you want to insert another Node <Y/N>... y
Enter Node Info
Enter Info : 20
Do you want to insert another Node <Y/N>... y
Enter Node Info
Enter Info : 30
Do you want to insert another Node <Y/N>... n
1. Insert
2. Delete
3. Traverse
4. Search
5. Update
Press 0 to Exit
Enter your Choice : 1
Enter Position : 2
Enter Info :40
10
40
20
30
Want to Continue <Y/N>...

```

## 2. Search

```
1. Insert
2. Delete
3. Traverse
4. Search
5. Update

Press 0 to Exit
Enter your Choice : 4

Enter Info : 20

Search Value found at 3 Position.
Want to Continue (Y/N)... _
```

## 3. Delete

```
1. Insert
2. Delete
3. Traverse
4. Search
5. Update

Press 0 to Exit
Enter your Choice : 2

Enter Position : 3

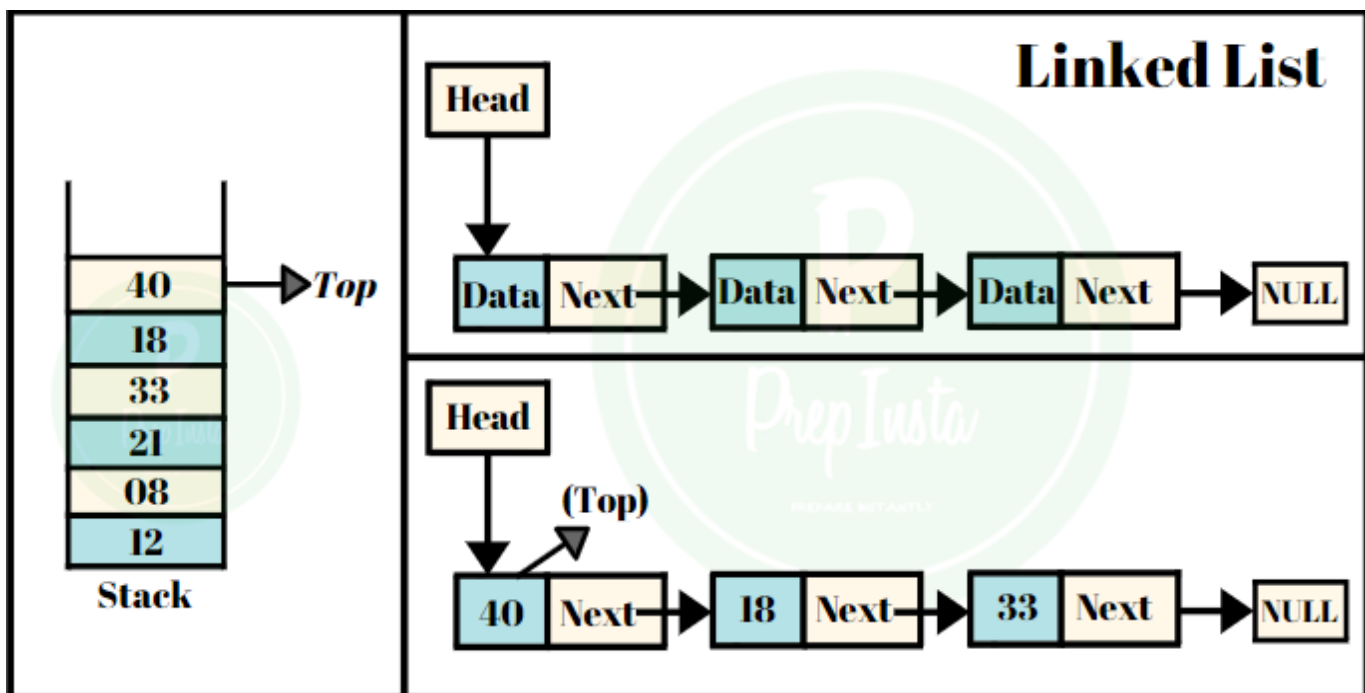
10
40
30
Want to Continue (Y/N)... _
```

# REPRESENTATION OF STACKS USING LINKED LISTS

A Linked List is a Data Structure which consists of two parts:

- The data/value part.
- The pointer which gives the location of the next node in the list.

Stack can also be represented by using a linked list. We know that in the case of arrays we face a limitation, i.e., array is a data structure of limited size. Hence before using an array to represent a stack we will have to consider an enough amount of space to suffice the memory required by the stack.



However, a Linked List is a much more flexible data structure. Both the push and pop operations can be performed on either ends.. But We prefer performing the Push and pop operation at the beginning.

The Stack operations occur in constant time. But if we perform the push and pop operations at the end of the linked list it takes time  $O(n)$ .

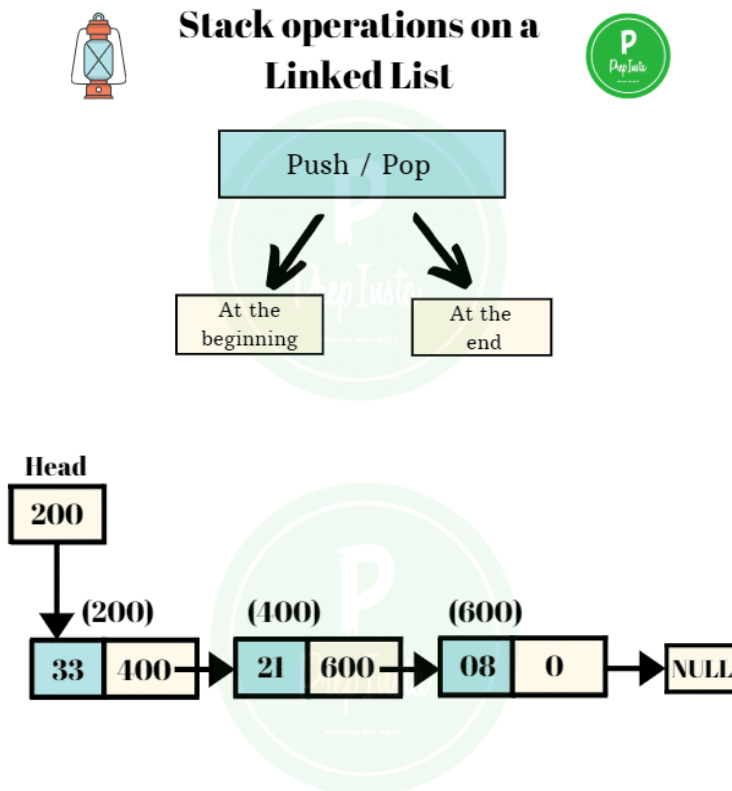
But performing the operations at the beginning occurs in constant time. Let us understand this with the help of an example.

Let us consider a linked list as shown here.



In the given data we can see that we have-

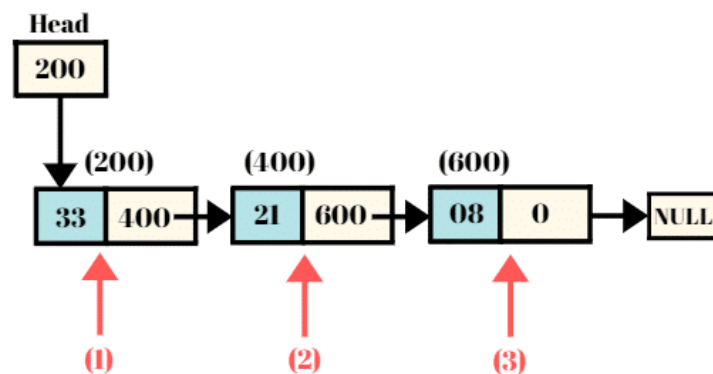
- **Head = 200.**
- **Top = 33.**



### Push / Pop At the end Of a Linked List

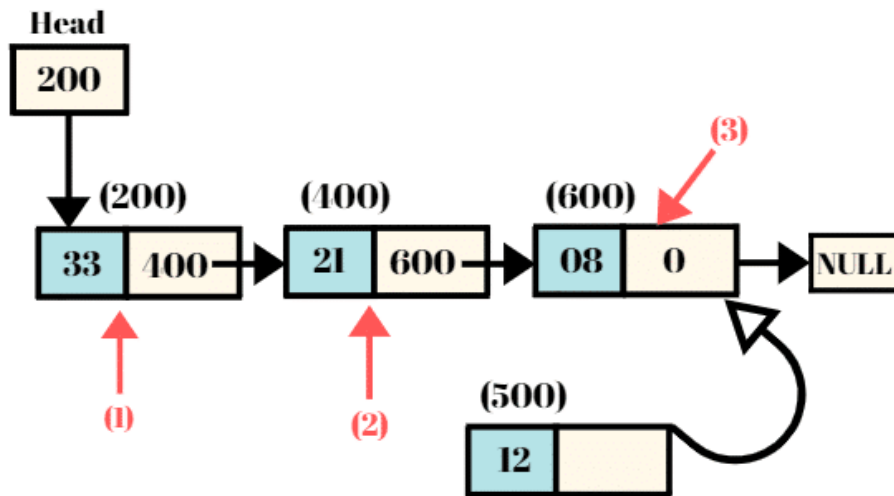
Consider the Linked List as shown above. Now if we want to push/pop a node at the end, we have to traverse all the  $n$  number of nodes, until we reach the end of the linked list.

We traverse the whole list from the beginning to the end.

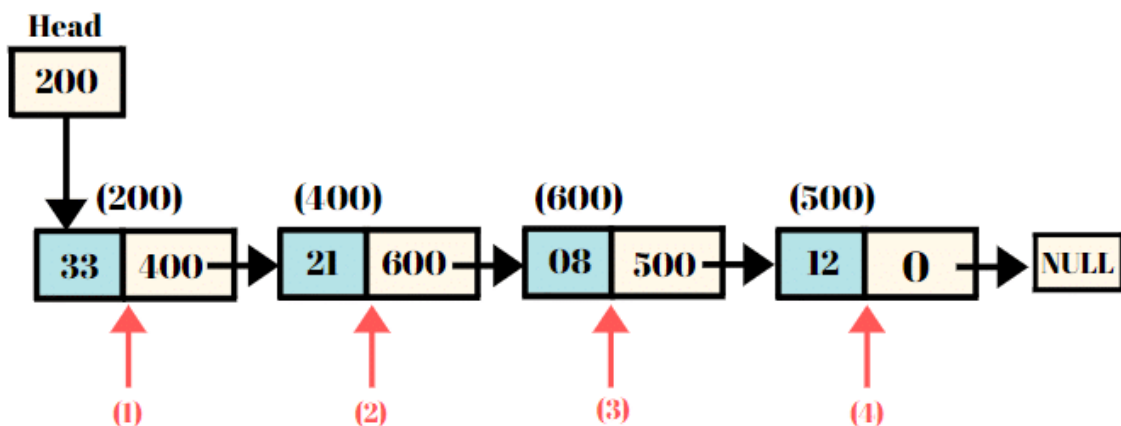


- Now if we want to insert a node at the end of the linked list, we traverse from the first node to the second node to the last node.

- We find the end of the list where the node points to the null and the node to be inserted is as shown here.



- The new node is pushed at the end of the list.
- The second last node , i.e, **08** now points to the location of the last node.
- The newly inserted node at the end now points to null.

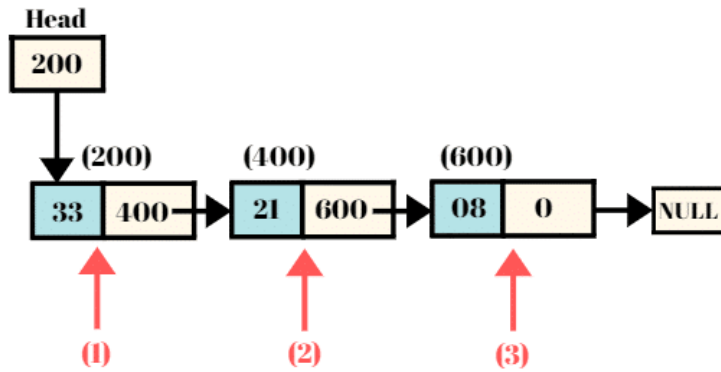


#### Push / Pop operation at the beginning of the Linked List

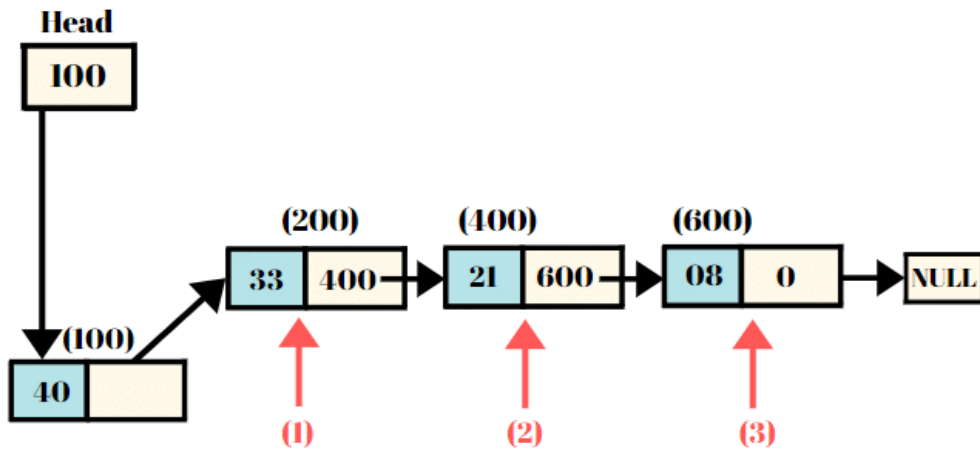
Pushing a new node at the beginning of a linked list takes place in constant time. When inserting the new node the number of traversals that occur is equal to 1. So it is much faster than the previous method of insertion

The time complexity of inserting a new node at the beginning of the linked list is  $O(1)$ .

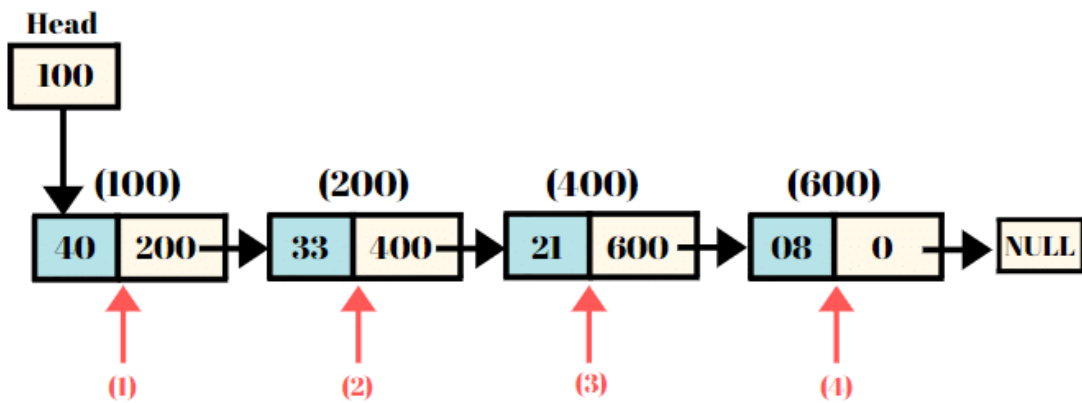
Let us consider the previous example only. We are given a Linked List as follows:



- A new node to be inserted to the Linked List is shown here.
- The new node 40 is to be inserted at the beginning of the list.



- The new node is inserted at the beginning where we have just a single traversal after spotting the head of the Linked List.
- The value of head changes from 200 to 100.
- The pointer of the new node stores the address of the next node.
- The final linked list is shown here.



# REPRESENTATION OF QUEUE USING LINKED LISTS

## Steps for implementing queue using linked list

### enqueue(data)

- Build a new node with given data.
- Check if the queue is empty or not.
- If queue is empty then assign new node to front and rear.
- Else make next of rear as new node and rear as new node.

### dequeue()

- Check if queue is empty or not.
- If queue is empty then dequeue is not possible.
- Else store front in temp
- And make next of front as front.

### print()

- Check if there is some data in the queue or not.
- If the queue is empty print "No data in the queue."
- Else define a node pointer and initialize it with front.
- Print data of node pointer until the next of node pointer becomes NULL.

