MODULE -2

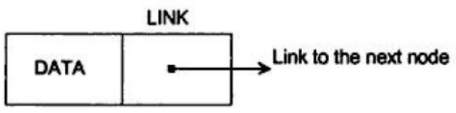
Linked Lists: Singly Linked List - Representation in Memory, Static, Dynamic - Operations on a Singly Linked List – Insertion, Deletion, Searching and Traversal- Types of Linked List- Circular Linked Lists, Doubly Linked Lists - Linked List Representation of Stack and Queue.

LINKED LIST

Array is a data structure where elements are stored in consecutive memory locations. In order to occupy the adjacent space, block of memory that is required for the array should be allocated before hand. Once memory is allocated it cannot be extended any more. This is why array is known as **static data structure**.

In contrast to this, linked list is called **dynamic data structure** where amount of memory required can be varied during its use. A linked list does not stores its elements in consecutive memory locations and the user can store any number of elements in it. However unlike an array a linked list does not allow random access of data. Elements in a linked list can be accessed only in a sequential manner. But insertions and deletions can be done at any point in the list.

In linked list, adjacency between the elements are maintained by means of links or pointers. A link or pointer actually is the address (memory location) of the next element. Thus, in a linked list, data (actual content) and link (to point to the next data) both are required to be maintained. An element in a linked list is specially termed as node, which can be viewed as

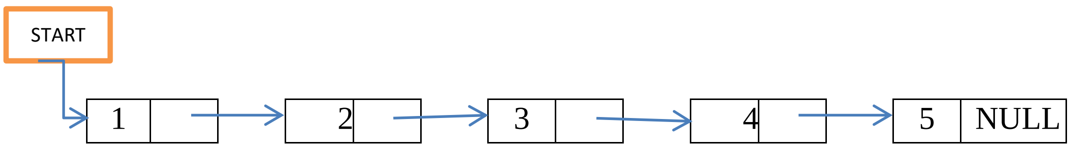


A node consists of two fields: DATA (to store the actual information) and LINK (to point to the next node).

Basic terminologies in a linked list

A linked list is a linear collection of data elements. These data elements are called **nodes.** Each node has one or more data fields and a pointer field to store the address of the next node.

Example:



In the above linked list every node has two fields: an integer field and a pointer field to store the address of the next node. The last node will have no next node connector to it. So pointer field of the last node will have a special value called NULL. Hence a NULL pointer denotes the end of the list.

Linked list contains a special pointer variable START which stores the address of first node in the list. This pointer is called **header**. We can traverse the entire list using START. If START is equal to NULL, then the linked list is empty and contains no nodes in the list.

<u>Representation of a Linked List in Memory</u>

There are two ways to represent a linked list in memory:

1. Static representation using array

2. Dynamic representation using free pool of storage

Static representation

Static representation of a single linked list maintains two arrays: one array for data and other for links.
<u>Example</u>

	Data	Link
0	H	4
1		
2		
3		
4	E	7
5		
6		
7	L	8
8	L	10
9		
10	0	NULL

Dynamic representation

The efficient way of representing a linked list is using free pool of storage. In this method, there is a memory bank (which is nothing but a collection of free memory spaces), and a memory manager (a program, in fact). During the creation of linked list, whenever a node is required the request is placed to the memory manager; memory manager will then search the memory bank for the block requested and if found grants a desired block to the caller. Again, there is also another program called garbage collector, it plays whenever a node is no more in use; it returns the unused node to the memory bank. Such a memory management is known as **dynamic memory management**. Dynamic representation of linked list uses the dynamic memory management policy.

<u>Linked Lists versus Arrays</u>

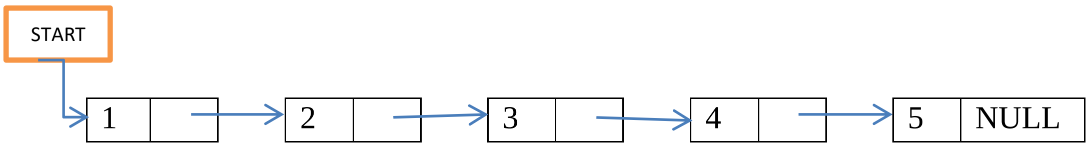
Both array and linked list are linear collection of elements. But unlike an array a linked list does not stores its nodes in consecutive memory location. A linked list does not allow random access of data. Nodes in a linked list can be accessed only in a sequential manner. We can add any number of elements in a linked list whereas in an array only limited number of elements can be inserted. For example, if we declare an array as int a[20]; then the array can have a maximum of 20 elements. There are no such restrictions in case of linked list. That means in an array memory allocation is static whereas in a linked list memory allocation is dynamic.

There are three types of linked lists:

1. Singly linked list.
2. Circular linked list.
3. Doubly linked list.

SINGLY LINKED LIST

A singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node. The pointer stores the address of the next node. The address of the first node is stored into a special pointer called '**START'**, also called **header**. Singly linked list allows traversal of data only in one direction from start to the end. The pointer field of the last node contains a special value called '**NULL**' which denotes the end of the list.



Representation of a node in a linked list

The node in a linked list can be represented using a structure.
For example, a node in the above linked list can be represented as:

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

Here a node contains two fields: an integer field and a pointer field of the type struct node which stores the address of next node.

Operations on a singly linked list

The following are different operations that can be performed on a singly linked list.

1. Traversal

2. Insertion
    a. Insertion at beginning of the list.
    b. Insertion at end of the list.
    c. Inserting a new node after a given node.
3. Deletion
    a. Deletion of first node.
    b. Deletion of last node.
    c. Deletion of a specific node.
4. Searching for a node.

Singly Linked List

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int data;
        struct node *next;
};
struct node *start=NULL;
void traversal();
void insertbegin ();
void insertend();
void insertafter();
void deletefirst();
void deletelast();
void deletespecific();
void search();
int main()
{
…………
}
```

1. **Traversal Operation**

Traversing a list means accessing the nodes of the list. For traversing, we use a pointer variable ‘ptr’ which points to the node that currently being accessed.

Algorithm

```
1. Set ptr = start
2. Repeat steps 3 and 4 while ptr!=NULL
3. Display ptr->data
4. Set ptr = ptr->next
5. Exit
```

Program

```
void traversal()
{

        struct node *ptr=start;

        if(start==NULL)
        {
                printf(“\nList is empty”);
        }
        else
        {
                printf(“\nThe elements are :”);
                while(ptr!=NULL)
                {
                        printf(“%d “,ptr->data);
                        ptr=ptr->next;
                }
        }
}
```

2. **<u>Insertion Operation</u>**

   There are three cases of insertion:
   a. insertion at beginning.
   b. insertion at the end.
   c. inserting a node after a given node.

For inserting a new node into a linked list, we need to allocate memory for the node dynamically. Also when we delete a node from a list, we need to dellocate the memory already allocated for the node.

<u>Dynamic Memory allocation and deallocation in C</u>

There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

1. malloc()
2. calloc()
3. free()
4. realloc()

1. **malloc() method**

The **“malloc”** or **“memory allocation”** method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn't Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

Syntax:
ptr = (cast-type*) malloc(byte-size)

For example,
int *ptr = (int*) malloc(sizeof(int));

The above statement allocates memory for an integer variable and sores its address to ptr.

int *ptr = (int*) malloc(100 * sizeof(int));

allocates 100 memory locations for storing integer values and assign address of the first location to ptr.

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

allocates memory for storing elements of type struct node.

2. **calloc() method**

"calloc" or **"contiguous allocation"** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:
1. It initializes each block with a default value '0'.
2. It has two parameters or arguments as compare to malloc().

Syntax:
ptr = (cast-type*)calloc(n, element-size);

here, n is the no. of elements and element-size is the size of each element.

For Example:

int *ptr = (int*) calloc(100, sizeof(int));

This statement allocates 100 memory locations for storing integer values and assign address of the first location to ptr.

3. **free() method**

"free" method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free()

method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax:

free(ptr);

4. **realloc() method**

“realloc” or **“re-allocation”** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. Re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

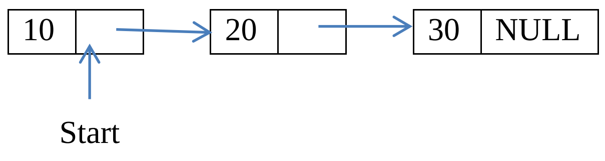
Syntax:

ptr = realloc(ptr, newSize);

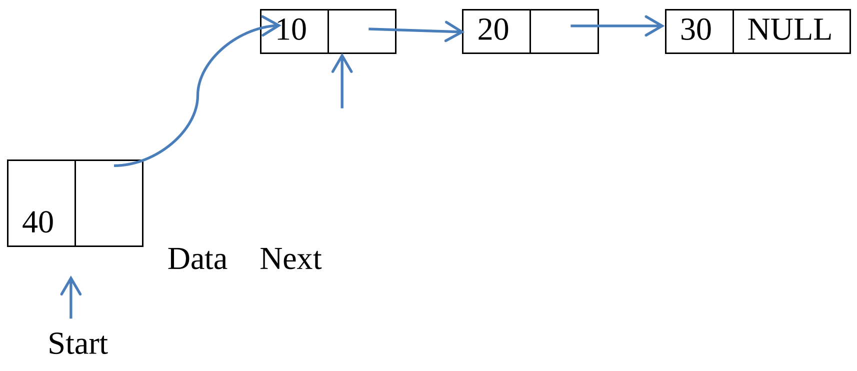
where ptr is reallocated with new size 'newSize'

a. <u>Insertion at beginning of the list.</u>

Suppose we want to add new node with data 40 at the beginning of the following list.



After insertion the list becomes:



<u>Algorithm</u>

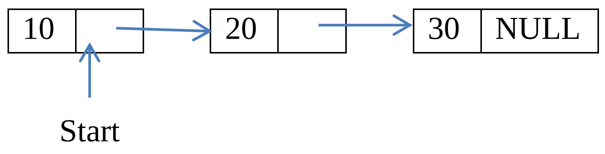
1. Allocate memory for the new node.
2. Store the new value in its data field.
3. Store the value of start in its next field..
4. Store the address of new node in start
5. Stop.

<u>Program</u>

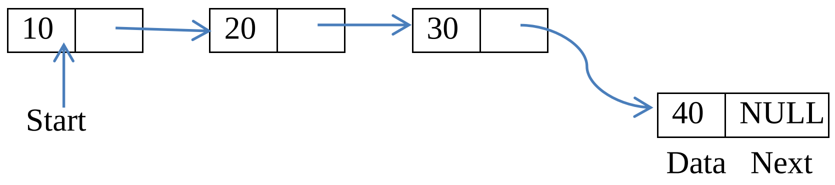
```
void insertbegin()
{
        int val;
        struct node *ptr;
        printf("\nEnter the value to be inserted");
        scanf("%d",&val);
        ptr=(struct node *)malloc(sizeof(struct node));
        ptr->data=val;
        ptr->next=start;
        start=ptr;
    }
```

b. <u>Insertion at end of the list</u>

Suppose we want to insert a new node with value 40 at end of the following list:



After insertion the list becomes:



<u>Algorithm</u>

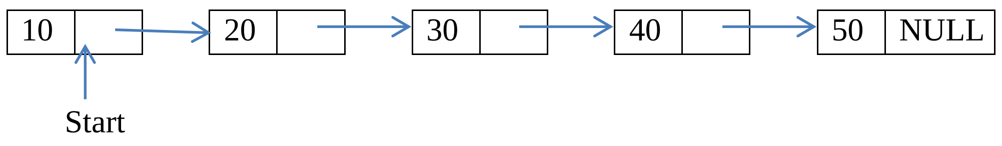
1. Allocate memory for the new node.
2. Store the new value in its data field and NULL in its next field
3. Traverse through the list from start to find the last node.
4. Once we reach the last node, store the address of new node to the next field of the last node.
5. Stop.

<u>Program</u>

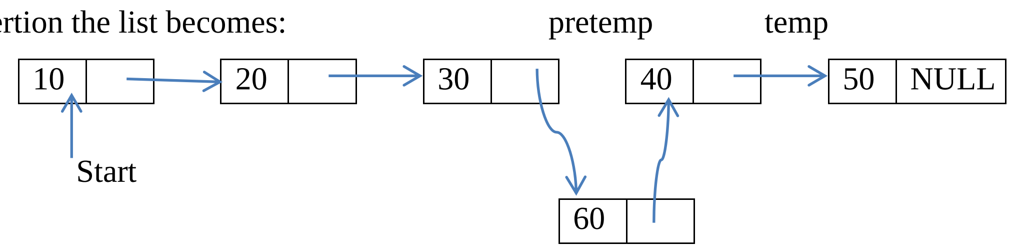
```
    void insertend()
    {
        int val;
        struct node *ptr, *temp;
        printf("\nEnter the value to be inserted");
        scanf("%d",&val);
        ptr=(struct node *)malloc(sizeof(struct node));
        ptr->data=val;
        ptr->next=NULL;
        temp=start;
        while(temp->next!=NULL)
            temp=temp->next;
            temp->next=ptr;
    }
```

c. Inserting a new node after a given node.

Suppose we want to insert a new node with value 60 after 30 in the following list:



After insertion the list becomes:



Algorithm

1. Allocate memory for the new node and store the new value in its data field.
2. Take two pointer variables 'pretemp' and 'temp' and initialize them with start and start->next respectively.
3. Move pretemp and temp until the data field of pretemp is equal to the value of the node after which insertion has to be done. pretemp will always point to the node just before temp.
4. Store the address of the new node to the next field of pretemp and store the address of temp to the next field of new node.
5. Stop.

Program

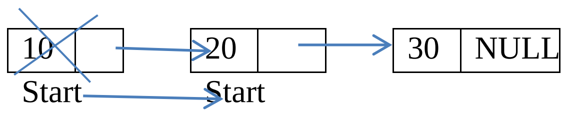
```
    void insertafter()
    {
            int val,item;
            struct node *pretemp,*temp, *ptr;
            printf("\nEnter the value to be inserted and the value after which it is
            inserted");
            scanf("%d%d",&val,&item);
            ptr=(struct node *)malloc(sizeof(struct node));
            ptr->data=val;
            pretemp=start;
            temp=start->next;
            while(pretemp->data !=item)
            {
                    pretemp = temp;
                    temp = temp->next;
            }
            pretemp->next = ptr;
            ptr->next = temp;
    }
```

3. Deletion Operation

There are three cases of deletion:

a. Deletion of first node.
b. Deletion of last node.
c. Deletion of a specific node.

a. Deletion of first struct node.



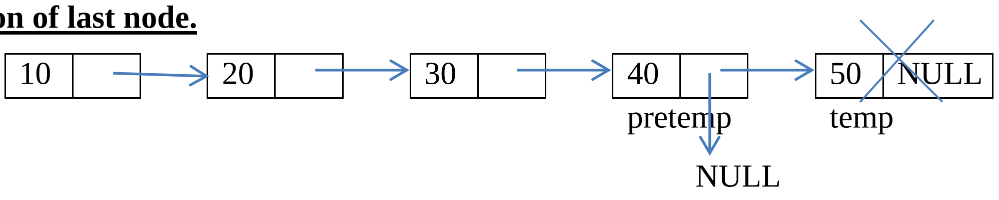
Algorithm

Step 1: Check whether the list is empty (if start == NULL). If so go to step 3.
Step 2: Make start to point to the next node (start = start->next) and de-allocate the memory allocated for the first node.
Step 3: Stop;

Program

```
void deletefirst()
{
        struct node *ptr;
        if (start == NULL)
                printf("The list is empty\n");
        else
        {
                ptr = start;
                start = start->next;
                printf("The deleted item is %d \n",ptr->data);
                free(ptr);
        }
}
```

b. Deletion of last node.



Algorithm

Step 1 : Check whether the list is empty (if start == NULL). If so the deletion is not possible and go to step 5. Otherwise if start->next=NULL, then the list will contain only one node, delete that node, set start=NULL and go to step 5.
Step2 :Take two pointer variables 'pretemp' and 'temp' and initialize them with start and start->next respectively.

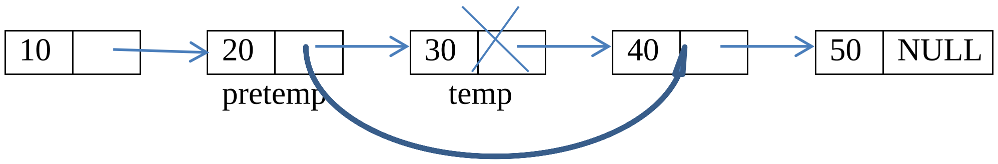
Step 3: Move pretemp and temp until the next part of temp becomes NULL. pretemp always points to the node just before temp.
Step 4: Set the next field of pretemp to NULL and delete the memory allocated for temp.
Step 5: Stop;

Program

```
void deletelast()
{
        struct node *pretemp,*temp;
        if (start == NULL)
                printf("The list is empty\n");
        else if(start->next == NULL)
        {
                printf("\nThe deleted item is: %d ",start->data);
                free(start);
                start = NULL;
        }
        else
        {
                pretemp= start;
                temp = start->next;
                while(temp->next !=NULL)
                {
                        pretemp = pretemp-next;
                        temp = temp->next;
                }
                pretemp->next = NULL;
                printf("\nThe deleted item is  : %d",temp->data);
                free(temp);
        }
}
```

c. **Deletion of a specified node**

Suppose we want to delete the node containing data 30 from the following linked list.



Algorithm

Step 1 : Check whether the list is empty (if start == NULL). If so the deletion is not possible and go to step 6.
Step 2: Check whether the first node contains the specified data. If so delete the first node and go to step 6.

Step3 :Take two pointer variables ‘pretemp’ and ‘temp’ and initialize them with start and start->next respectively.
Step 4: Move pretemp and temp until temp contains either the specified data or becomes NULL. If temp becomes NULL, the specified data is not in the list and go to step 6. pretemp always points to the node just before temp.
Step 5: Set the next field of pretemp to the next field of temp (node after temp) and then deallocate the memory allocated for temp.
Step 6: Stop;

<u>Program</u>

```
void deletespecific()
{
        int val;
        struct node *pretemp, *temp;
        printf(“\nEnter the value that is to be deleted”):
        scanf(“%d”,&val);
        if (start == NULL)
                printf(“\nThe list is empty”);
        else if(start->data == val)
                deletefirst();
        else
        {
                pretemp= start;
                temp = start->next;
                while((temp->data !=val)&&(temp!=NULL))
                {
                        pretemp = pretemp-next;
                        temp = temp->next;
                }
                if(temp==NULL)
                        printf(“\nNo node with data %d is present in the list”,val);
                else
                {
                        pretemp->next = temp->next;
                        free(temp);
                }
        }
}
```

4. **<u>Searching operation</u>**
   Searching means finding whether a given value is present in the data field of any node or not.

<u>Algorithm</u>

```
    Step 1: Set ptr = start
```

```
Step 2: Repeat step 3 until ptr =NULL
Step 3: if ptr->data = val, go to step 4
        else set ptr = ptr->next
Step 4: if ptr!=NULL the given value is present in the list, otherwise the value searched is not in the list.
Step 5: Stop
```

<u>Program</u>

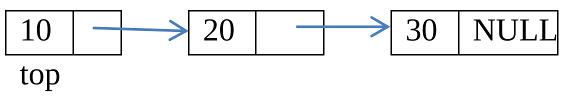
```
void search()
{
    int val;
    struct node *ptr;
    printf("\nEnter the data to be searched:");
    scanf("%d",&val);
    ptr = start;
    while(ptr!=NULL)
    {
            if(ptr->data == val)
            {
                    printf("\n Element Found");
                    break;
            }
            else
                    ptr = ptr->next;
    }
    if(ptr ==NULL)
            printf("\n Element Not Found");
}
```

<u>LINKED REPRESENTATION OF STACK (LINKED STACK)</u>

In stack, insertions and deletions are performed at one end called its top. The drawback of array representation of stack is that the array must be declared to have some fixed size. But the linked list can be dynamically grow.

In a linked stack every node has two parts – one that stores data and another that stores the address of next node. The start pointer, which stores the address of the first node, is used as top of the stack. All insertions and deletions are done at the node pointed by top. If top is equal to NULL then it indicates that the stack is empty.

<u>Example of a linked stack:</u>



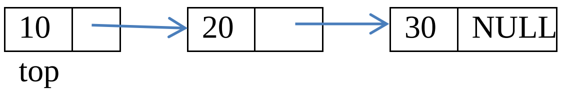
The operations that can be performed on stack are push, pop and traversal.

Linked Stack

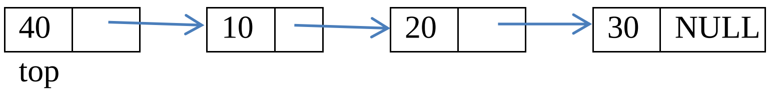
```
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int data;
        struct node *next;
};
struct node *top=NULL;
void push();
void pop();
void traversal();
int main()
{
…………
}
```

Push Operation

The push operation is used to insert an element in the stack. The new element is added at the top position of the stack. Suppose we want to insert an element with value 40 in the following linked stack.



We insert the new node at the beginning of the stack and make the pointer top to point to the new node. Thus after insertion, the stack becomes:



Algorithm

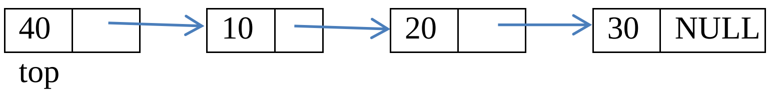
```
Step 1: Allocate memory for the new node and assign its address to ptr;
Step 2: Set ptr->data = value
Step 3: Set ptr->next = top
Step 4: Set top = ptr
Step 5: Stop
```

Program

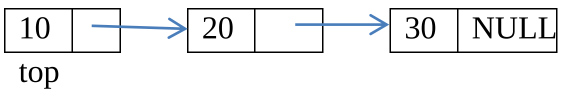
```
void push()
{
        int val;
        struct node *ptr;
        printf(“\nEnter the value to be inserted: “);
        scanf(“%d”,&val);
        ptr= (struct node *)malloc(sizeof(struct node));
        ptr->data = val;
        ptr->next = top;
        top = ptr;
}
```

Pop Operation

Pop operation is used to delete the topmost element from a stack. Before deleting the value, we have to check whether top=NULL. If so, stack is empty. Otherwise, delete the node pointed by top and make top to point to the next node. For example, consider the following stack:



After pop operation, the stack will be as:



<u>Algorithm</u>

```
Step 1: If top = NULL, stack will be empty and go to step 5
Step 2: Set ptr = top
Step 3: Set top = top->next
Step 4: Deallocate the memory allocated for ptr
Step 5: Stop
```

<u>Program</u>

```
void pop()
{
        struct node *ptr;
        if(top == NULL)
                printf(“\n Stack Underflow”);
        else
        {
                ptr=top;
                printf(“\nThe element to be deleted is : %d“,top->data);
                top = ptr->next;
                free(ptr);
        }
```

}

Traversal Operation

Traversal operation is same as that of a singly linked list.

Algorithm

```
Step 1: If top = NULL, stack will be empty and go to step 4
Step 2: Set ptr = top and repeat step3 until ptr=NULL
Step 3: Display ptr->data and Set ptr = ptr->next
Step 4: Stop
```

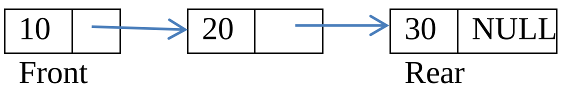
Program

```
void traversal()
{
        struct node *ptr;
        if(top == NULL)
                printf("\n Stack Underflow");
        else
        {
                ptr= top;
                printf("\nThe elements are : ");
                while(ptr!=NULL)
                {
                        printf("%d ",ptr->data);
                        ptr = ptr->next;
                }
        }
}
```

LINKED REPRESENTATION OF QUEUE (LINKED QUEUE)

In linked queue every element has two parts, one that stores data and another that stores the address of next element. The start pointer of the linked list is used as front and the rear pointer will store the address of last element in the queue. All insertions will be done at the rear end and all deletion will be done at the front end. If front = rear =NULL then it indicates that queue is empty.

Example of a linked queue:

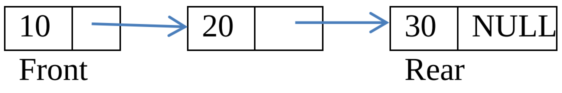


The operations that can be performed on a queue are insertion, deletion and traversal.

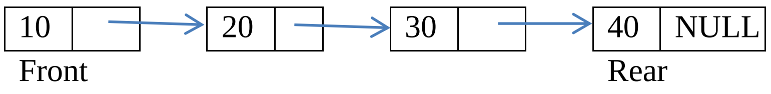
```
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int data;
        struct node *next;
};
struct node *front=NULL, *rear=NULL;

void insertion();
void deletion();
void traversal();
int main()
{
…….
}
```

Insertion Operation

Insertion operation inserts an element at the rear end of the queue. Consider the following linked queue.



Suppose we want to insert an element with value 40. The new node is inserted at the end of the queue and store its address to rear pointer. After insertion the queue becomes:



If the new node is inserted into an empty queue (ie, if front=rear=NULL), just set front and rear pointers as the address of the new node.

Algorithm

```
Step 1: Allocate memory for the new node and initialize a pointer variable ptr with its address.
Step 2 : Set ptr->data = val;
        Set ptr->next = NULL;
Step 3: If front == NULL, set front=rear=ptr;
        Otherwise,      Set rear->next = ptr;
                        Set rear=ptr;
Step 4: Stop
```

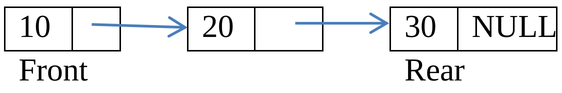
Program

void insertion()

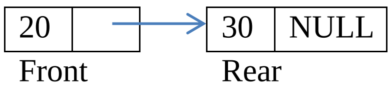
```
{
        int val;
        printf("\nEnter the value to be inserted: ");
        scanf("%d",&val);
        struct node *ptr = (struct node *)malloc(sizeof(struct node));
        ptr->data = val;
        ptr->next = NULL;
        if(front == NULL)
        {
                front = ptr;
                rear = ptr;
        }
        else
        {
                rear->next = ptr;
                rear = ptr;
        }

}
```

<u>Deletion Operation</u>

Deletion operation deletes the element pointed by front from the queue. If front == NULL then the queue is empty and no more deletions can be done. After deletion, front will point to the second element. For example, consider the following queue:



After deletion, queue becomes:



<u>Algorithm</u>

Step 1: If front ==NULL, Queue is empty and go to step 5.
Step 2 : Set ptr = front
Step 3: If front == rear, Set front=rear=NULL, otherwise, Set front = front->next
Step 4: Deallocate the memory allocated for ptr (delete ptr)
Step 5: Stop

<u>Program</u>

```
void deletion()
{
        struct node *ptr;
        if(front == NULL)
                printf("\nQueue empty");
        else
        {
                ptr= front;
                printf("\n The element deleted is : %d",ptr->data);
                free(ptr);
                if (front == rear)
                {
                        front = NULL;
                        rear = NULL;
                }
                else
                        front = front->next;
        }

}
```

<u>Traversal Operation</u>

Traversal is same as that of singly linked list.

<u>Algorithm</u>

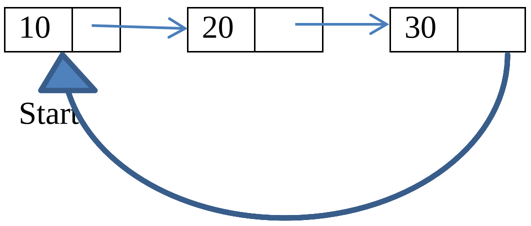
```
Step 1: If front = NULL, queue will be empty and go to step 4
Step 2: Set ptr = front and repeat step 3 until ptr=NULL
Step 3: Display ptr->data and Set ptr = ptr->next
Step 4: Stop
```

<u>Program</u>

```
void linkedstack::traversal()
{
        struct node *ptr;
        if(front == NULL)
                printf("\n Queue Underflow");
        else
        {
                ptr= front;
                printf("\nThe elements are : ");
                while(ptr!=NULL)
                {
                        printf("%d ",ptr->data);
                        ptr = ptr->next;
                }
        }
}
```

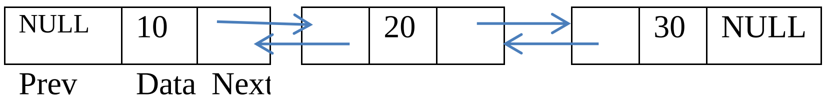
CIRCULAR LINKED LIST

In a circular linked list, last node points to the first node of the list.



There is no NULL value in the pointer field of last node. We can traverse the list from first node until we find a node whose next field contains the address of first node. This denotes the end of the circular list. Here the node which contains the address of first is actually the last node.

DOUBLY LINKED LIST OR TWO – WAY LINKED LIST



Each node contains a pointer to the next as well as the previous node. Therefore each node has three parts: data, a pointer to the next node and a pointer to the previous node. Thus a node can be represented as:

```
struct node
{
        struct node *prev;
        int data;
        struct node *next;
};
```

The previous field of the first node and next field of the last node will contain NULL value.

Advantage: We can traverse the list in both direction (forward as well as backward)
Disadvantage: Memory space required for a node is more than that of a singly linked list.