Chapter – 3

# QUEUE AND LINKED LIST

## 1. Queue

### 1.1 Definition and Queue Operations

- A Queue is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.
- Queue, like Stack, is also an abstract data structure. The thing that makes queue different from stack is that a queue is open at both its ends.
- The data is inserted into the queue through one end and deleted from it using the other end.
- A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.
- For example, people waiting in line for a rail ticket form a queue.

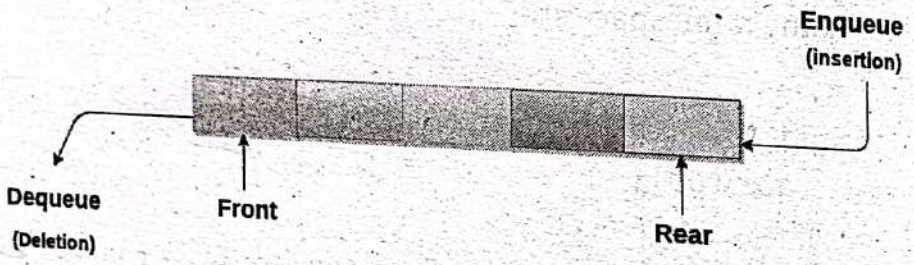



*Fig.: Queue data structure*

**Basic Operations**

- Queue operations also include initialization of a queue, usage and permanently deleting the data from the memory.
- The most fundamental operations in the queue ADT include: enqueue(), dequeue(), peek(), isFull(), isEmpty().
- These are all built-in operations to carry out data manipulation and to check the status of the queue.
- Queue uses two pointers – **front** and **rear**. The front pointer accesses the data from the front end (helping in enqueueing) while the rear pointer accesses data from the rear end (helping in dequeuing).
- **enqueue()** – Insertion of elements to the queue.
- **dequeue()** – Removal of elements from the queue.

- **peek() or front():** Acquires the data element available at the front node of the queue without deleting it.
- **rear():** This operation returns the element at the rear end without removing it.
- **isFull():** Validates if the queue is full.
- **isEmpty():** Checks if the queue is empty.
- **size():** This operation returns the size of the queue i.e. the total number of elements it contains.

## Applications of Queue

Due to the fact that queue performs actions on a first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queues are used to maintain the playlist in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.

## Primitive operation in Queue [2013 Spring]

## Algorithm for Enqueue operation in Queue

In this algorithm, a 1-D array(Q) with size 'N' is Q[N] being used as a queue with variables front and rear to keep track of both the ends and elements 'x' is added to the queue if it is not already full!!!.

**Step I:** check if (rear>=N-1)

```
{
   Display "Queue is full!!!"
   exit
}
 else
{
   If (front==-1)
   front=0;
}
```

**Step II:** set rear=rear+1;

**Step III:** set Q[rear]=x;

**Step IV:** end

### Algorithm for dequeue operation in Queue

In this algorithm, a 1-D array(Q) with size 'N' is Q[N] being used as a queue with variables front and rear to keep track of both the ends and elements 'x' is added to the queue if it is not already full!!!.

**Step I:** check if (front=rear=-1 or front>rear)

```
{
    Display "Queue is empty!!!"
    exit
}
```

**Step II:** set x=Q[front];

**Step III:** set front=front+1;

**Step IV:** end/stop

## 1.2 Queue ADT and its Array Implementation

### Queue as an ADT

A queue q of type T is a finite sequence of elements with the operations:

- **MakeEmpty(q):** To make q as an empty queue
- **IsEmpty(q):** To check whether the queue q is empty. Return true if q is empty, return false otherwise.
- **IsFull(q):** To check whether the queue q is full. Return true in q is full, return false otherwise.
- **Enqueue(q, x):** To insert an item x at the rear of the queue, if and only if q is not full.
- **Dequeue(q):** To delete an item from the front of the queue q. if and only if q is not empty.
- **Traverse (q):** To read the entire queue that displays the content of the queue.

Thus by using a queue we can perform above operations thus a queue acts as an ADT.

### Array implementation using queue [2014 Fall]

- **To implement a queue using an array,**
  - create an array arr of size **n** and
  - take two variables **front** and **rear** both of which will be initialised to 0 which means the queue is currently empty.

- **Element**
  - rear is the index up to which the elements are stored in the array and
  - front is the index of the first element of the array.

**Below is the implementation of a queue using an array:**

- In the below code , we are initialising front and rear as 0, but in general we have to initialise it with -1.

  If we assign rear as 0, rear will always point to next block of the end element, in fact , rear should point the index of last element,

  eg. When we insert element in queue , it will add in the end i.e. after the current rear and then point the rear to the new element,

  **According to the following code:**

  IN the first dry run, front=rear = 0;

  - in void queueEnqueue(int data)
  - else part will be executed,
  - so arr[rear] =data;// rear =0, rear pointing to the latest element
  - rear++; //now rear = 1, rear pointing to the next block after end element not the end element
  - //that's against the original definition of rear

**// C++ program to implement a queue using an array**

```
#include <bits/stdc++.h>
using namespace std;

struct Queue {
  int front, rear, capacity;
  int* queue;
  Queue(int c)
  {
    front = rear = 0;
    capacity = c;
    queue = new int;
  }

  ~Queue() { delete[] queue; }

  // function to insert an element
  // at the rear of the queue
```

```
void queueEnqueue(int data)
{
  // check queue is full or not
  if (capacity == rear) {
    printf("\nQueue is full\n");
    return;
  }

  // insert element at the rear
  else {
    queue[rear] = data;
    rear++;
  }
  return;
}

// function to delete an element
// from the front of the queue
void queueDequeue()
{
  // if queue is empty
  if (front == rear) {
    printf("\nQueue is  empty\n");
    return;
  }

  // shift all the elements from index 2 till rear
  // to the left by one
  else {
    for (int i = 0; i < rear - 1; i++) {
      queue[i] = queue[i + 1];
    }

    // decrement rear
    rear--;
  }
  return;
```

```
    }

    // print queue elements
    void queueDisplay()
    {
      int i;
      if (front == rear) {
        printf("\nQueue is Empty\n");
        return;
      }

      // traverse front to rear and print elements
      for (i = front; i < rear; i++) {
        printf(" %d <-- ", queue[i]);
      }
      return;
    }

    // print front of queue
    void queueFront()
    {
      if (front == rear) {
        printf("\nQueue is Empty\n");
        return;
      }
      printf("\nFront Element is: %d", queue[front]);
      return;
    }
};

// Driver code
int main(void)
{
  // Create a queue of capacity 4
  Queue q(4);

  // print Queue elements
```

```
q.queueDisplay();

// inserting elements in the queue
q.queueEnqueue(20);
q.queueEnqueue(30);
q.queueEnqueue(40);
q.queueEnqueue(50);

// print Queue elements
q.queueDisplay();

// insert element in the queue
q.queueEnqueue(60);

// print Queue elements
q.queueDisplay();

q.queueDequeue();
q.queueDequeue();

printf("\n\nafter two node deletion\n\n");

// print Queue elements
q.queueDisplay();

// print front of the queue
q.queueFront();

return 0;
}
```

Output

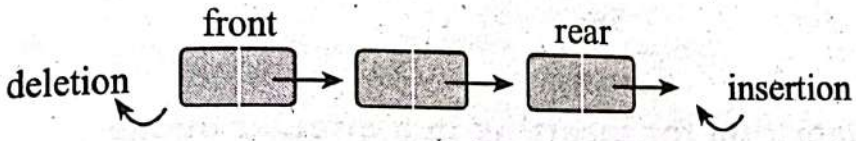
```
Queue is Empty
20 <-- 30 <-- 40 <-- 50 <--
Queue is full
20 <-- 30 <-- 40 <-- 50 <--
after two node deletion
40 <-- 50 <--
Front Element is: 40
```

**Types of Queue** [VVI]

1. Linear Queue/simple queue
2. Circular queue
3. Double ended queue (deque)
4. Priority queue

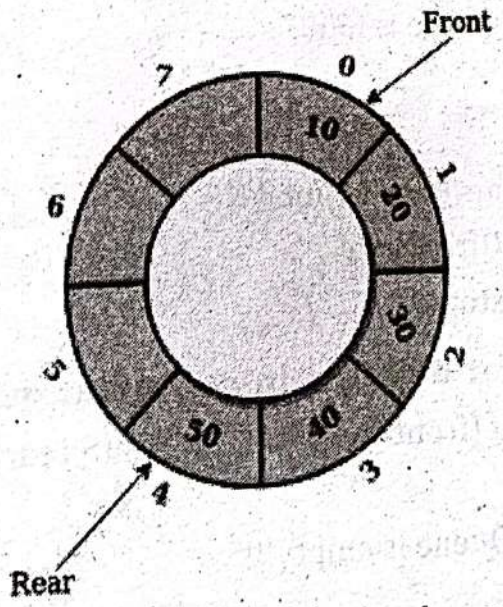
### 1. Linear Queue/ Simple Queue

A Queue is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order. In a simple queue, insertion takes place at the rear and removal occurs at the front. It strictly follows the FIFO (First in First out) rule.




### 2. Circular Queue

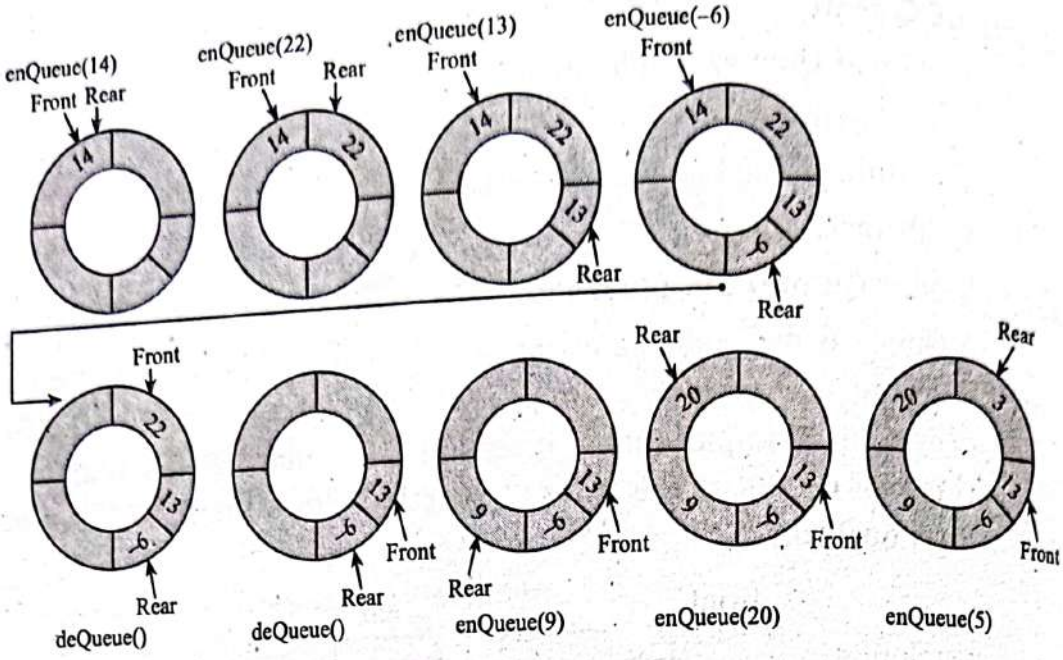
- A Circular Queue is an extended version of a normal queue where the last element of the queue is connected to the first element of the queue forming a circle.
- The operations are performed based on FIFO (First In First Out) principle. It is also called **'Ring Buffer'**.




In a normal Queue, we can insert elements until the queue becomes full. But once the queue becomes full, we can not insert the next element even if there is a space in front of the queue.

### Illustration of Circular Queue Operations:

Follow the below image for a better understanding of the enqueue and dequeue operations.




### Algorithm for inserting in a circular queue [VVI]

In this algorithm 1-D array CQ[N] is being used with variables front and rear as pointers to keep track of both the ends. An element 'x' is added into a circular queue of size (N) if it is already not full.

**Step I:** check if ((rear+1)mod N==front)

{

Display "Queue is full!!!"

exit

}

**Step II:** set rear=(rear+1)mod N;

**Step III:** set CQ[rear]=x;

**Step IV:** end/stop

### Algorithm for deleting in a circular queue

**Step I:** check if (front=rear=N-1 or front> rear)

{

Display "Queue is empty!!!"

exit

}

**Step II:** set x=CQ[front];

**Step III:** set front=(front+1) mod N;

**Step IV:** end/stop

### Implement Circular Queue using Array:

1. Initialise an array queue of size **n**, where n is the maximum number of elements that the queue can hold.

2. Initialize two variables front and rear to -1.
3. **Enqueue:** To enqueue an element **x** into the queue, do the following:
   a. Increment rear by 1.
      If **rear** is equal to n, set **rear** to 0.
   b. If **front** is -1, set **front** to 0.
   c. Set queue[rear] to x.
4. **Dequeue:** To dequeue an element from the queue, do the following:
   a. Check if the queue is empty by checking if **front** is -1.
      If it is, return an error message indicating that the queue is empty.
   b. Set **x** to queue [front].
   c. If **front** is equal to **rear**, set **front** and **rear** to –1.
   d. Otherwise, increment **front** by 1 and if **front** is equal to n, set **front** to 0.
   e. Return x.

### Implementation of circular queue using an Array

[2019 Spring]

```
#include <stdio.h>
# define max 6
int queue[max];  // array declaration
int front=-1;
int rear=-1;
// function to insert an element in a circular queue
void enqueue(int element)
{
  if(front==-1 && rear==-1) // condition to check queue is empty
  {
    front=0;
    rear=0;
    queue[rear]=element;
  }
  else if((rear+1)%max==front)// condition to check queue is full
  {
    printf("Queue is overflow..");
  }
  else
  {
```

```
    rear=(rear+1)%max;      // rear is incremented
    queue[rear]=element;    // assigning a value to the queue at
    the rear position.
  }
}

// function to delete the element from the queue
int dequeue()
{
  if((front==-1) && (rear==-1))  // condition to check queue is
    empty
  {
    printf("\nQueue is underflow..");
  }
 else if(front==rear)
 {
  printf("\nThe dequeued element is %d", queue[front]);
  front=-1;
  rear=-1;
 }
 else
 {
  printf("\nThe dequeued element is %d", queue[front]);
  front=(front+1)%max;
 }
 }
 // function to display the elements of a queue
 void display()
 {
   int i=front;
   if(front==-1 && rear==-1)
   {
     printf("\n Queue is empty..");
   }
   else
   {
     printf("\nElements in a Queue are :");
```

```
        while(i<=rear)
        {
          printf("%d,", queue[i]);
          i=(i+1)%max;
        }
      }
    }
    int main()
    {
      int choice=1,x;  // variables declaration

      while(choice<4 && choice!=0)  // while loop
      {
      printf("\n Press 1: Insert an element");
      printf("\nPress 2: Delete an element");
      printf("\nPress 3: Display the element");
      printf("\nEnter your choice");
      scanf("%d", &choice);

      switch(choice)
      {
        case 1:

        printf("Enter the element which is to be inserted");
        scanf("%d", &x);
        enqueue(x);
        break;
        case 2:
        dequeue();
        break;
        case 3:
        display();
     }}
      return 0;
    }
```

## Benefits of Circular Queue over linear queue

[2013 Spring/2017 Fall]

- **Efficient Memory Usage:** In a linear queue, when elements are dequeued, the space at the front becomes unused and is wasted. Circular queues, on the other hand, reuse this space as the front pointer wraps around to the beginning when it reaches the end of the array, allowing for more efficient memory usage.
- **Better Utilisation of Resources:** Circular queues are particularly useful in scenarios where there is a fixed amount of memory available. They help in better resource utilisation by recycling the space that becomes available after dequeuing elements.
- **Simplified Implementation:** Implementing a circular queue is often simpler than a linear queue. The circular nature allows for more straightforward handling of the pointers without the need for special cases when the front or rear pointer reaches the end of the array.
- **Avoiding Shifting of Elements:** In a linear queue, when an element is dequeued, all the remaining elements may need to be shifted to fill the gap. Circular queues eliminate the need for shifting because the front pointer can simply move to the next position.
- **Efficient for Certain Applications:** Circular queues are commonly used in scenarios where there is a need to continuously process a stream of data or in situations where a fixed-size buffer is used, such as in networking applications, keyboard buffers, and real-time systems.
- **Faster Enqueue and Dequeue Operations:** Pointers wrap around without additional checks, leading to faster operations.

## Real-time Applications of Circular Queue:

- **Months in a year:** Jan –> Feb –> March –> and so on upto Dec–>Jan –> .....
- **Eating:** Breakfast –> lunch –> snacks –> dinner –> breakfast –> and so on .....
- Traffic Light is also a real-time application of a circular queue.
- Clock is also a better example for the Circular Queue.

## Advantages of Circular Queue:

- It provides a quick way to store FIFO data with a maximum size.
- Efficient utilisation of the memory.
- Doesn't use dynamic memory.

- Simple implementation.
- All operations occur in O(1) constant time.

**Disadvantages of Circular Queue:** [2019 Fall]

- In a circular queue, the number of elements you can store is only as much as the queue length, you have to know the maximum size beforehand.
- Some operations like deletion or insertion can be complex in circular queue.
- The implementation of some algorithms like priority queue can be difficult in a circular queue.
- Circular queue has a fixed size, and when it is full, there is a risk of overflow if not managed properly.
- In the array implementation of Circular Queue, even though it has space to insert the elements it shows that the Circular Queue is full and gives the wrong size in some cases.

**Algorithm for traversing in a circular queue**

1. Initialise a variable 'current' to the front of the circular queue.
2. If the circular queue is empty, print an appropriate message and exit.
3. While 'current' is not equal to the rear of the circular queue, do the following:
   a. Print the element at the 'current' position.
   b. Move 'current' to the next position in the circular queue. (Use modulo operation to handle wrap-around at the end of the queue)
4. Print the element at the 'current' position (the last element in the circular queue).

## 3. Double ended queue

- The deque stands for Double Ended Queue. Deque is a linear data structure where the insertion and deletion operations are performed from both ends. We can say that deque is a generalised version of the queue.
- Though the insertion and deletion in a deque can be performed on both ends, it does not follow the FIFO rule. The representation of a deque is given as follows:


Representation of deque

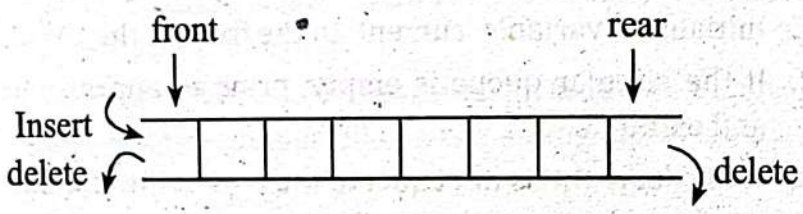
**Types of deque**

There are two types of deque:

- Input restricted queue
- Output restricted queue

i. **Input restricted deque**

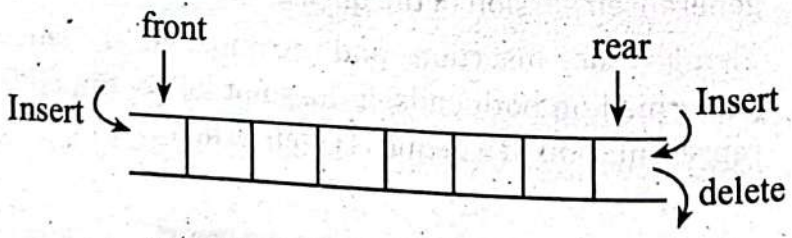
- In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.
- In this type of Queue, the input can be taken from one side only(rear) and deletion of elements can be done from both sides(front and rear).
- This kind of Queue does not follow FIFO(first in first out).
- This queue is used in cases where the consumption of the data needs to be in FIFO order but if there is a need to remove the recently inserted data for some reason and one such case can be irrelevant data, performance issue, etc.




input restricted double ended queue

ii. **Output restricted deque**

- In the output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.
- In this type of Queue, the input can be taken from both sides(rear and front) and the deletion of the element can be done from only one side(front).
- This queue is used in the case where the inputs have some priority order to be executed and the input can be placed even in the first place so that it is executed first.




output restricted double ended queue

## Double ended queue algorithm (deque Algorithm) [2015 Spring]

### Algorithm for Insertion at rear end

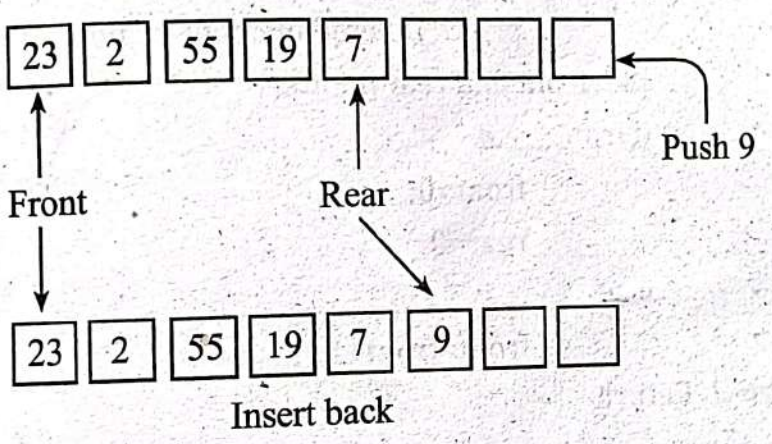
**Step-1:** [Check for overflow]

```
if(rear==MAX)
        Print("Queue is Overflow");
        return;
```

**Step-2:** [Insert Element]

```
else
        rear=rear+1;
        q[rear]=no;
[Set rear and front pointer]
if rear=0
        rear=1;
if front=0
        front=1;
```

**Step-3:** return




### Algorithm for Insertion at front end

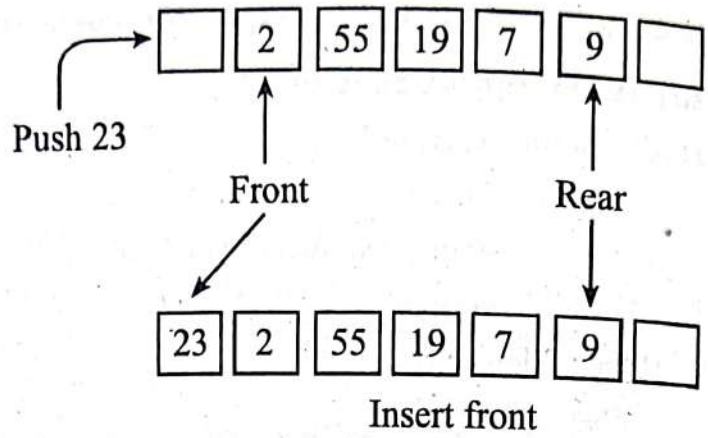
**Step-1:** [Check for the front position]

```
if(front<=1)
        Print("Cannot add item at the front");
        return;
```

**Step-2:** [Insert at front]

```
else
        front=front-1;
        q[front]=no;
```

**Step-3:** Return




**Algorithm for Deletion from front end**

**Step-1** [ Check for front pointer]

if front=0

print(" Queue is Underflow");

return;

**Step-2** [Perform deletion]

else

no=q[front];

print("Deleted element is",no);

[Set front and rear pointer]

if front=rear

front=0;

rear=0;

else

front=front+1;

**Step-3:** Return

**Algorithm for Deletion from rear end**

**Step-1:** [Check for the rear pointer]

if rear=0

print("Cannot delete value at rear end");

return;

**Step-2:** [ perform deletion]

else

no=q[rear];

[Check for the front and rear pointer]

if front= rear

front=0;

rear=0;

```
        else
                rear=rear-1;
                print("Deleted element is",no);
```

**Step-3:** Return

## Implementation of Deque using an Array

```
#include <stdio.h>
#define size 5
int deque[size];
int f = -1, r = -1;
// insert_front function will insert the value from the front
void insert_front(int x)
{
  if((f==0 && r==size-1) || (f==r+1))
  {
    printf("Overflow");
  }
  else if((f==-1) && (r==-1))
  {
    f=r=0;
    deque[f]=x;
  }
  else if(f==0)
  {
    f=size-1;
    deque[f]=x;
  }
  else
  {
    f=f-1;
    deque[f]=x;
  }
}

// insert_rear function will insert the value from the rear
void insert_rear(int x)
{
  if((f==0 && r==size-1) || (f==r+1))
  {
    printf("Overflow");
```

```
    }
    else if((f==-1) && (r==-1))
    {
      r=0;
      deque[r]=x;
    }
    else if(r==size-1)
    {
      r=0;
      deque[r]=x;
    }
    else
    {
      r++;
      deque[r]=x;
    }

}

// display function prints all the values of deque.
void display()
{
  int i=f;
  printf("\nElements in a deque are: ");

  while(i!=r)
  {
    printf("%d ",deque[i]);
    i=(i+1)%size;
  }
  printf("%d",deque[r]);
}

// getfront function retrieves the first value of the deque.
void getfront()
{
  if((f==-1) && (r==-1))
  {
    printf("Deque is empty");
```

```
    }
    else
    {
      printf("\nThe value of the element at front is: %d", deque[f]);
    }

}

// getrear function retrieves the last value of the deque.
void getrear()
{
  if((f==-1) && (r==-1))
  {
    printf("Deque is empty");
  }
  else
  {
    printf("\nThe value of the element at rear is %d", deque[r]);
  }

}

// delete_front() function deletes the element from the front
void delete_front()
{
  if((f==-1) && (r==-1))
  {
    printf("Deque is empty");
  }
  else if(f==r)
  {
    printf("\nThe deleted element is %d", deque[f]);
    f=-1;
    r=-1;

  }
  else if(f==(size-1))
```

```
    {
      printf("\nThe deleted element is %d", deque[f]);
      f=0;
    }
    else
    {
      printf("\nThe deleted element is %d", deque[f]);
      f=f+1;
    }
}

// delete_rear() function deletes the element from the rear
void delete_rear()
{
  if((f==-1) && (r==-1))
  {
    printf("Deque is empty");
  }
  else if(f==r)
  {
    printf("\nThe deleted element is %d", deque[r]);
    f=-1;
    r=-1;

  }
  else if(r==0)
  {
    printf("\nThe deleted element is %d", deque[r]);
    r=size-1;
  }
  else
  {
    printf("\nThe deleted element is %d", deque[r]);
    r=r-1;
  }
}
```

```
int main()
{
  insert_front(20);
  insert_front(10);
  insert_rear(30);
  insert_rear(50);
  insert_rear(80);
  display(); // Calling the display function to retrieve the values
    of deque
  getfront(); // Retrieve the value at front-end
  getrear(); // Retrieve the value at rear-end
  delete_front();
  delete_rear();
  display(); // calling display function to retrieve values after
    deletion
  return 0;
}
```

## 4. Priority Queue

- A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.
- The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.
- For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

### Characteristics of a Priority queue

A priority queue is an extension of a queue that contains the following characteristics:

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.

- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

**Let's understand the priority queue through an example.**

We have a priority queue that contains the following values:

1, 3, 4, 8, 14, 22

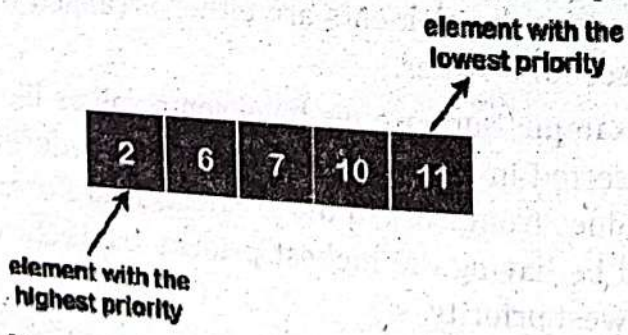
All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

- **poll():** This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
- **add(2):** This function will insert the '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
- **poll():** It will remove the '2' element from the priority queue as it has the highest priority queue.
- **add(5):** It will insert 5 elements after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

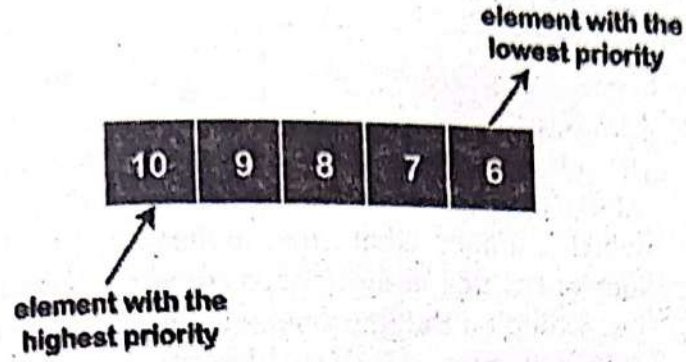
**Types of Priority Queue**

**There are two types of priority queue:**

- **Ascending order priority queue:** In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.

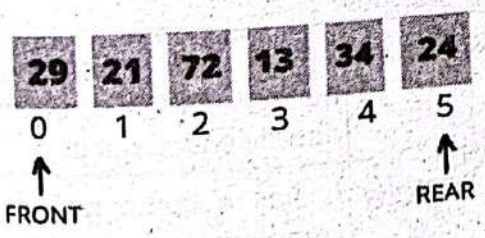



- **Descending order priority queue:** In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e.5 is given as the highest priority in the queue.

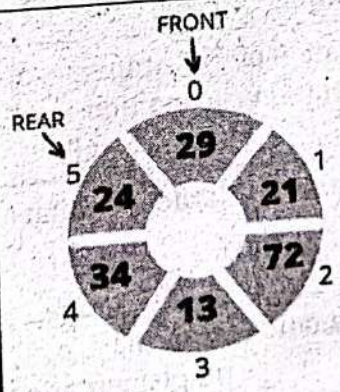



**Differentiate Linear Queue vs circular queue.** [2014 Spring]

| Basis of comparison | Linear Queue | Circular Queue |
|---|---|---|
| Meaning | The linear queue is a type of linear data structure that contains the elements in a sequential manner. | The circular queue is also a linear data structure in which the last element of the Queue is connected to the first element, thus creating a circle. |
| Insertion and Deletion | In a linear queue, insertion is done from the rear end, and deletion is done from the front end. | In a circular queue, the insertion and deletion can take place from any end. |
| Memory space | The memory space occupied by the linear queue is more than the circular queue. | It requires less memory as compared to a linear queue. |
| Memory utilisation | The usage of memory is inefficient. | The memory can be more efficiently utilised. |
| Order of execution | It follows the FIFO principle in order to perform the tasks. | It has no specific order for execution. |




**LINEAR QUEUE**




**CIRCULAR QUEUE**

**Stack vs Queue** [2015 Fall]

| Basis for comparison | Stack | Queue |
|---|---|---|
| Principle | It follows the principle LIFO (Last In- First Out), which implies that the element which is inserted last would be the first one to be deleted. | It follows the principle FIFO (First In -First Out), which implies that the element which is added first would be the first element to be removed from the list. |
| Structure | It has only one end from which both the insertion and deletion take place, and that end is known as a top. | It has two ends, i.e., front and rear end. The front end is used for the deletion while the rear end is used for the insertion. |
| Number of pointers used | It contains only one pointer known as a top pointer. The top pointer holds the address of the last inserted or the topmost element of the stack. | It contains two pointers: front and rear pointer. The front pointer holds the address of the first element, whereas the rear pointer holds the address of the last element in a queue. |
| Operations performed | It performs two operations, push and pop. The push operation inserts the element in a list while the pop operation removes the element from the list. | It performs mainly two operations, enqueue and dequeue. The enqueue operation performs the insertion of the elements in a queue while the dequeue operation performs the deletion of the elements from the queue. |
| Examination of the empty condition | If top==-1, which means that the stack is empty. | If front== -1 or front = rear+1, which means that the queue is empty. |
| Examination of full condition | If top== max-1, this condition implies that the stack is full. | If rear==max-1, this condition implies that the stack is full. |
| Variants | It does not have any types. | It is of three types like priority queue, circular queue and double ended queue. |
| Implementation | It has a simpler implementation. | It has a comparatively more complex implementation than a stack. |
| Visualisation | A Stack is visualised as a vertical collection. | A Queue is visualised as a horizontal collection. |

## 2. Linked List

### 2.1 List Definition and its Operation

- The list can be defined as an abstract data type in which the elements are stored in an ordered manner for easier and efficient retrieval of the elements.
- List Data Structure allows repetition that means a single piece of data can occur more than once in a list.
- In the case of multiple entries of the same data, each entry of that repeating data is considered as a distinct item or entry.
- It is very much similar to the array but the major difference between the array and the list data structure is that array stores only homogenous data in them whereas the list (in some programming languages) can store heterogeneous data items in its object. List Data Structure is also known as a sequence.
- The list can be called Dynamic size arrays, which means their size increases as we go on adding data in them and we need not to pre-define a static size for the list.
- In computer science and data structures, a list is a collection of elements, where each element typically holds some data and a reference (or link) to the next element in the sequence.
- Lists are versatile data structures used to organise and store data in a linear fashion.
- There are various types of lists, including arrays, linked lists, doubly linked lists, and circular linked lists, each with its own set of characteristics and advantages.

**Here are some common list operations in data structures:**

- **Access (Indexing):** Retrieving the value of an element at a specific position (index) in the list.
- **Insertion:** Adding a new element to the list. This operation can involve inserting an element at the beginning, end, or a specific position within the list.
- **Deletion:** Removing an element from the list. Similar to insertion, deletion can occur at the beginning, end, or a specific position within the list.
- **Traversal:** Visiting each element in the list one by one. This is often done using loops to perform operations on each element.
- **Search:** Finding the position or existence of a specific element within the list.
- **Concatenation:** Combining two or more lists to create a new list.

- **Splitting:** Dividing a list into two or more separate lists.
- **Sorting:** Arranging the elements of the list in a specific order, such as ascending or descending.
- **Merging:** Combining two sorted lists into a single sorted list.
- **Reversal:** Inverting the order of elements in the list.
- **Length (Size):** Determining the number of elements in the list.
- **Empty Check:** Checking if the list is empty or not.

**Static list vs dynamic list**

| Static list | Dynamic list |
|---|---|
| Fixed size | Variable size |
| Size determined at compile-time | Size is determined as run time |
| Memory allocation in compile time | Memory allocation in run time. |
| Can't grow | Can grow or shrink in runtime |
| Generally, more memory efficient | More flexible but memory-efficient may involve additional overhead. |
| Generally array in the programming | Linked list, arraylist, vectors. |

## 2.2 List ADT and its Array Implementation

**List Abstract Data Type (ADT):**

**Operations:**

**MakeEmpty:** Create an empty list.

**IsEmpty:** Check if the list is empty.

**IsFull:** Check if the list is full (if there is a maximum capacity).

**Length:** Return the number of elements in the list.

**Insert:** Add an element to the list at a specified position.

**Delete:** Remove an element from the list at a specified position.

**Retrieve:** Return the value of an element at a specified position.

**Find:** Find the position of a specified element in the list.

**MakeEmpty:** Remove all elements from the list, leaving it empty.

**Next:** Return the position of the element following a specified element in the list.

**Previous:** Return the position of the element preceding a specified element in the list.

**First:** Return the position of the first element in the list.

**Last:** Return the position of the last element in the list.

**Print:** Output the elements of the list.

## Array implementation of List

**Construction:**

**Step 1:** Start the program

**Step 2:** Initialise and declare variables using structure and arrays. Define the required size of header files

**Step 3:** Enter the operations to perform in the list as

a. Create a list
b. Insert
c. Delete
d. View

**Step 4:** Based on the operations chosen, the list elements are structured.

**Step 5:** Stop the program

**Program:**

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define LIST_SIZE 30
void main()
{
int *element=NULL;
int ch,i,j,n;
int insdata,deldata,moddata,found;
int top=-1;
element=(int*)malloc(sizeof(int)* LIST_SIZE);
clrscr();
while(1)
{
fflush(stdin);
printf("\n\n basic Operations in a Linear List.....");
printf("\n 1.Create New List \t 2.Modify List \t 3.View List");
printf("\n 4.Insert First \t 5.Insert Last \t 6.Insert Middle");
printf("\n 7.Delete First \t 8.Delete Last \t 9.Delete Middle");
printf("\nEnter the Choice 1 to 10 : ");
scanf("%d",&ch);
```

```
switch(ch)
{
case 1:
top=-1;
printf("\n Enter the Limit (How many Elements):");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("\n Enter The Element [%d]:",(i+1));
scanf("%d",&element[++top]);
}
break;
case 2:
if(top==-1)
{
printf("\n Linear List is Empty:");
break;
}
printf("\n Enter the Element for Modification:");
scanf("%d",&moddata);
found=0;
for(i=0;i<=top;i++)
{
if(element[i]==moddata)
{
found=1;
printf("\n Enter The New Element :");
scanf("%d",&element[i]);
break;
}
}
if(found==0)
printf("\n Element %d not found",moddata);
break;
case 3:
if(top==-1)
printf("\n \n Linear List is Empty:");
```

```
else if(top==LIST_SIZE -1)
printf("\n Linear LIst is Full:");
for(i=0;i<=top;i++)
printf("\n Element[%d]is-->%d",(i+1),element[i]);
break;
case 4:
if(top==LIST_SIZE-1)
{
printf("\n Linear List is Full:");
break;
}
top++;
for(i=top;i>0;i--)
element[i]=element[i-1];
printf("\n Enter the Element:");
scanf("%d",&element[0]);
break;
case 5:
if(top==LIST_SIZE-1)
{
printf("\n Linear List is Full:");
break;
}
printf("\n Enter the Element:");
scanf("%d",&element[++top]);
break;
case 6:
if(top==LIST_SIZE-1)
printf("\n Linear List is Full:");
else if(top==-1)
printf("\n linear List is Empty.");
else
{
found=0;
printf("\n Enter the Element after which the insertion is to be
    made:");
scanf("%d",&insdata);
```

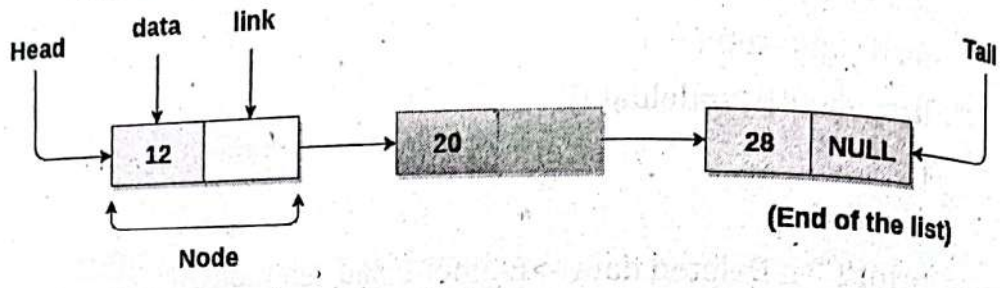
```
for(i=0;i<=top;i++)
if(element[i]==insdata)
{
found=1;
top++;
for(j=top;j>i;j--)
element[j]=element[j-1];
printf("\n Enter the Element :");
scanf("%d",&element[i+1]);
break;
}
if(found==0)
printf("\n Element %d Not Found",insdata);
}
break;
case 7:
if(top==-1)
{
printf("\n Linear List is Empty:");
break;
}
printf("\n Deleted Data-->Element :%d",element[0]);
top--;
for(i=0;i<=top;i++)
element[i]=element[i+1];
break;
case 8:
if(top==-1)
printf("\n Linear List is Empty:");
else
printf("\n Deleted Data-->Element :%d",element[top--]);
break;
case 9:
if(top==-1)
{
printf("\n Linear List is Empty:");
break;
```

```
}
printf("\n Enter the Element for Deletion :");
scanf("%d",&deldata);
found=0;
for(i=0;i<=top;i++)
if(element[i]==deldata)
{
found=1;
printf("\n Deleted data-->Element :%d",element[i]);
top--;
for(j=i;j<=top;j++)
element[j]=element[j+1];
break;
}
if(found==0)
printf("\n Element %d Not Found ",deldata);
break;
default:
free(element);
printf("\n End Of Run Of Your Program.....");
exit(0);
}
}
}
```

## 2.3 Linked List- Definition and Operation

- A linked list is a linear data structure that stores a collection of data elements dynamically.
- Nodes represent those data elements, and links or pointers connect each node.
- Each node consists of two fields, the information stored in a linked list and a pointer that stores the address of its next node.
- The last node contains null in its second field because it will point to no node.
- A linked list can grow and shrink its size, as per the requirement.
- It does not waste memory space.
- Linked List can be defined as a collection of objects called nodes that are randomly stored in the memory.

- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains a pointer to the null.




## Basic Operations in Linked List

The basic operations in the linked lists are insertion, deletion, searching, display, and deleting an element at a given key. These operations are performed on Singly Linked Lists as given below:

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

### Advantages of Linked Lists:

- **Dynamic Size:** Linked lists can easily grow or shrink in size during runtime, as memory allocation is dynamic. This flexibility is particularly useful when the number of elements is not known in advance.
- **Constant-Time Insertions and Deletions:** Insertions and deletions at any position in a linked list can be done in constant time O(1) if the position is given. This is in contrast to arrays, where inserting or deleting elements in the middle may require shifting elements, resulting in O(n) time complexity.
- **No Pre-allocation of Memory:** Linked lists do not require pre-allocation of memory for a specific size, unlike arrays. This can be advantageous in situations where the size of the data is uncertain.
- **Efficient Memory Utilisation:** Linked lists can minimise memory wastage by allocating exactly the required amount of memory for each element. In contrast, arrays may need to allocate space for a fixed-size block, leading to potential wasted memory.

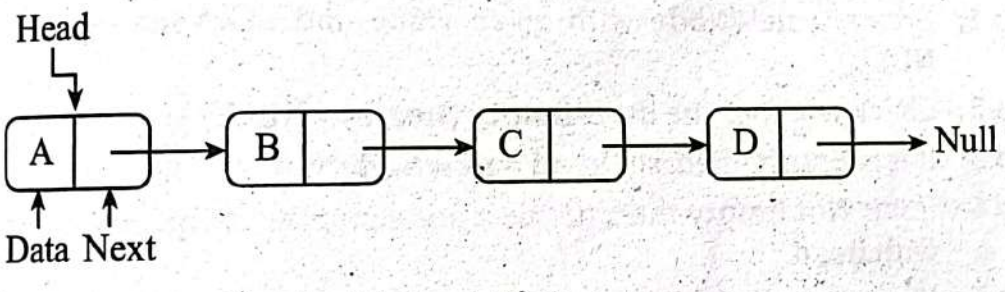
### Disadvantages of Linked Lists:

- **Random Access is Inefficient:** Unlike arrays, linked lists do not provide constant-time random access to elements. Accessing an element requires traversing the list from the beginning, resulting in O(n) time complexity.

- **Memory Overhead:** Each element in a linked list requires additional memory for the link (pointer/reference) to the next element. This can result in higher memory overhead compared to arrays, especially for small data types. On a 32-bit CPU, an extra 4 bytes per element are typically used to store these references.
- **Sequential Access:** Sequential access is generally more efficient in linked lists. However, if there is a need for frequent random access or index-based access, arrays may be more suitable.
- **Each Element is a Separate Object:** In a linked list, each element is considered a separate object with its own memory allocation and a reference to the next node. This can lead to a less compact representation in memory compared to arrays, where elements are stored contiguously.

## 2.4 Singly Linked List (SLL)

A **singly linked list** is a linear data structure in which the elements are not stored in contiguous memory locations and each element is connected only to its next element using a pointer.




### Operations on Single Linked List

The following operations are performed on a Single Linked List

- Insertion
- Deletion
- Display

Before we implement actual operations, first we need to set up an empty list. First, perform the following steps before implementing actual operations.

**Step 1:** Include all the **header files** which are used in the program.

**Step 2:** Declare all the **user defined functions.**

**Step 3:** Define a **Node** structure with two members **data** and **next**

**Step 4:** Define a Node pointer **'head'** and set it to **NULL.**

**Step 5:** Implement the main method by displaying the operations menu and make suitable function calls in the main method to perform user selected operation.

### Insertion

In a single 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 the beginning of the single linked list...

**Step 1:** Create a **newNode** with a given value.

**Step 2:** Check whether list is **Empty (head == NULL)**

**Step 3:** If it is **Empty** then, set **newNode → next = NULL** and **head = newNode.**

**Step 4:** If it is **Not Empty** then, set **newNode → next = head** and **head = newNode.**

### Inserting At End of the list

We can use the following steps to insert a new node at the end of the single linked list...

**Step 1:** Create a **newNode** with given value and **newNode → next** as **NULL.**

**Step 2:** Check whether the list is **Empty (head == NULL).**

**Step 3:** If it is **Empty** then, set **head = newNode.**

**Step 4:** If it is **Not Empty** then, define a node pointer **temp** and initialise with **head.**

**Step 5:** Keep moving the **temp** to its next node until it reaches the last node in the list (until **temp → next** is equal to **NULL**).

**Step 6:** Set **temp → next = newNode.**

### 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 single linked list...

**Step 1:** Create a **newNode** with a given value.

**Step 2:** Check whether list is **Empty (head == NULL)**

**Step 3:** If it is **Empty** then, set **newNode → next = NULL** and **head = newNode.**

**Step 4:** If it is **Not Empty** then, define a node pointer **temp** and initialise with **head.**

**Step 5:** Keep moving the **temp** to its next node until it reaches 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** has reached the last node or not. If it is reached to the last node then display **'Given node is**

**not found in the list!!! Insertion is not possible!!!'** and terminate the function. Otherwise move the **temp** to the next node.

**Step 7:** Finally, Set **'newNode → next = temp → next'** and **'temp → next = newNode'**

### Deletion

In a single linked list, the deletion operation can be performed in three ways. They 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 the beginning of the single 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 terminates the function.

**Step 3:** If it is **Not Empty** then, define a Node pointer **'temp'** and initialise with **head.**

**Step 4:** Check whether list is having only one node (**temp → next == NULL**)

**Step 5:** If it is **TRUE** then set **head = NULL** and delete **temp** (Setting **Empty** list conditions)

**Step 6:** If it is **FALSE** then set **head = temp → next**, and delete **temp.**

### Deleting from End of the list

We can use the following steps to delete a node from the end of the single 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 terminates the function.

**Step 3:** If it is **Not Empty** then, define two Node pointers **'temp1'** and **'temp2'** and initialise **'temp1'** with **head.**

**Step 4:** Check whether list has only one Node (**temp1 → next == NULL**)

**Step 5:** If it is **TRUE.** Then, set **head = NULL** and delete **temp1.** And terminate 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 it reaches the last node in the list. (until **temp1 → next == NULL**)

**Step 7:** Finally, Set **temp2 → next = NULL** and delete **temp1**.

### Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single 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 terminates the function.

**Step 3:** If it is **Not Empty** then, define two Node pointers **'temp1'** and **'temp2'** and initialise **'temp1'** with **head**.

**Step 4:** Keep moving the **temp1** until it reaches 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 is 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 or not

**Step 7:** If the 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 the 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 move the **head** to the next node **(head = head → next)** and delete **temp1**.

**Step 10:** If **temp1** is not the first node then check whether it is the last node in the list **(temp1 → next == NULL)**.

**Step 11:** If **temp1** is the last node then set **temp2 → next = NULL** and delete **temp1 (free(temp1))**.

**Step 12:** If **temp1** is not the first node and not the last node then set **temp2 → next = temp1 → next** and delete **temp1 (free(temp1))**.

### Displaying a Single Linked List

We can use the following steps to display the elements of a single 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 initialise 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 an arrow pointing to **NULL** (**temp → data → NULL**).

## Implementation of Single Linked List using C Programming

```
#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 wan to
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();
                    break;
              case 3:    printf("Enter the value which you want o
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;
      head = newNode;
    }
    printf("\nOne node inserted!!!\n");
  }
  void insertAtEnd(int value)
  {
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if(head == NULL)
      head = newNode;
    else
    {
      struct Node *temp = head;
      while(temp->next != NULL)
      temp = temp->next;
      temp->next = newNode;
    }
    printf("\nOne node inserted!!!\n");
  }
```

```
void insertBetween(int value, int loc1, int loc2)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = value;
  if(head == NULL)
  {
    newNode->next = NULL;
    head = newNode;
  }
  else
  {
    struct Node *temp = head;
    while(temp->data != loc1 && temp->data != loc2)
     temp = temp->next;
    newNode->next = temp->next;
    temp->next = newNode;
  }
  printf("\nOne node inserted!!!\n");
}
void removeBeginning()
{
  if(head == NULL)
    printf("\n\nList is Empty!!!");
  else
  {
    struct Node *temp = head;
    if(head->next == NULL)
    {
     head = NULL;
     free(temp);
    }
    else
    {
    head = temp->next;
    free(temp);
    printf("\nOne node deleted!!!\n\n");
```

```
    }
   }
  }
  void removeEnd()
  {
   if(head == NULL)
   {
    printf("\nList is Empty!!!\n");
   }
   else
   {
    struct Node *temp1 = head,*temp2;
    if(head->next == NULL)
     head = NULL;
    else
    {
     while(temp1->next != NULL)
     {
      temp2 = temp1;
      temp1 = temp1->next;
     }
     temp2->next = NULL;
    }
    free(temp1);
    printf("\nOne node deleted!!!\n\n");
   }
  }
  void removeSpecific(int delValue)
  {
   struct Node *temp1 = head, *temp2;
   while(temp1->data != delValue)
   {
    if(temp1 -> next == NULL){
     printf("\nGiven node not found in the list!!!");
     goto functionEnd;
    }
    temp2 = temp1;
    temp1 = temp1 -> next;
```

```
    }
    temp2 -> next = temp1 -> next;
    free(temp1);
    printf("\nOne node deleted!!!\n\n");
    functionEnd:
  }
  void display()
  {
    if(head == NULL)
    {
      printf("\nList is Empty\n");
    }
    else
    {
      struct Node *temp = head;
      printf("\n\nList elements are - \n");
      while(temp->next != NULL)
      {
       printf("%d --->",temp->data);
       temp = temp->next;
      }
      printf("%d --->NULL",temp->data);    }}
```

**Traversing in singly linked list**

Traversing is the most common operation that is performed in almost every scenario of a singly linked list. Traversing means visiting each node of the list once in order to perform some operation on that. This will be done by using the following statements.

```
ptr = head;
    while (ptr!=NULL)
        {
          ptr = ptr -> next;
        }
```

**Algorithm:**

**Step 1:** SET PTR = HEAD

**Step 2:** IF PTR = NULL
WRITE "EMPTY LIST"
GOTO STEP 7
END OF IF

**Step 4:** REPEAT STEP 5 AND 6 UNTIL PTR != NULL
**Step 5:** PRINT PTR→ DATA
**Step 6:** PTR = PTR → NEXT
**Step 7:** [END OF LOOP]
**Step 7:** EXIT

**C function**

```
#include<stdio.h>
#include<stdlib.h>
void create(int);
void traverse();
struct node
{
  int data;
  struct node *next;
};
struct node *head;
void main ()
{
  int choice,item;
  do
  {
printf("\n1.Append List\n2.Traverse\n3.Exit\n4.Enter your
choice?");
    scanf("%d",&choice);
    switch(choice)
    {
      case 1:
      printf("\nEnter the item\n");
      scanf("%d",&item);
      create(item);
      break;
      case 2:
      traverse();
      break;
      case 3:
      exit(0);
      break;
```

```
        default:
        printf("\nPlease enter valid choice\n");
      }

    } while(choice != 3);
}
void create(int item)
  {
   struct node *ptr = (struct node *)malloc(sizeof(struct node *));
    if(ptr == NULL)
    {
      printf("\nOVERFLOW\n");
    }
    else
    {
      ptr->data = item;
      ptr->next = head;
      head = ptr;
      printf("\nNode inserted\n");
    }
  }
void traverse()
  {
    struct node *ptr;
    ptr = head;
    if(ptr == NULL)
    {
      printf("Empty list..");
    }
    else
    {
      printf("printing values . . . . .\n");
      while (ptr!=NULL)
      {
        printf("\n%d",ptr->data);
        ptr = ptr -> next;
      } } }
```

## Searching in singly linked list

Searching is performed in order to find the location of a particular element in the list. Searching any element in the list needs traversing through the list and making the comparison of every element of the list with the specified element. If the element is matched with any of the list elements then the location of the element is returned from the function.

### Algorithm

**Step 1:** SET PTR = HEAD

**Step 2:** Set I = 0

**Step 3:** IF PTR = NULL
WRITE "EMPTY LIST"
GOTO STEP 8
END OF IF**STEP 4:** REPEAT STEP 5 TO 7 UNTIL PTR != NULL

**Step 5:** if ptr → data = item
write i+1
End of IF**STEP 6:** I = I + 1

**Step 7:** PTR = PTR → NEXT
[END OF LOOP]

**Step 8:** EXIT

### C function

```
#include<stdio.h>
#include<stdlib.h>
void create(int);
void search();
struct node
{
  int data;
  struct node *next;
};
struct node *head;
void main ()
{
  int choice,item,loc;
  do
  {
printf("\n1.Create\n2.Search\n3.Exit\n4.Enter your choice?");
    scanf("%d",&choice);
```

```
        switch(choice)
        {
          case 1:
          printf("\nEnter the item\n");
          scanf("%d",&item);
          create(item);
          break;
          case 2:
          search();
          case 3:
          exit(0);
          break;
          default:
          printf("\nPlease enter valid choice\n");
        }

      }while(choice != 3);
    }
      void create(int item)
      {
    struct node *ptr = (struct node *)malloc(sizeof(struct node *));
        if(ptr == NULL)
        {
          printf("\nOVERFLOW\n");
        }
        else
        {
          ptr->data = item;
    ptr->next = head;
          head = ptr;
          printf("\nNode inserted\n");
        }

      }
    void search()
    {
      struct node *ptr;
```

```
int item,i=0,flag;
ptr = head;
if(ptr == NULL)
{
  printf("\nEmpty List\n");
}
else
{
  printf("\nEnter item which you want to search?\n");
  scanf("%d",&item);
  while (ptr!=NULL)
  {
    if(ptr->data == item)
    {
      printf("item found at location %d ",i+1);
      flag=0;
    }
    else
    {
      flag=1;
    }
    i++;
    ptr = ptr -> next;
  }
  if(flag==1)
  {
    printf("Item not found\n");
  }
 }
}
```

**Doubly linked list**

- Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence.
- Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer).
- A sample node in a doubly linked list is shown in the figure.

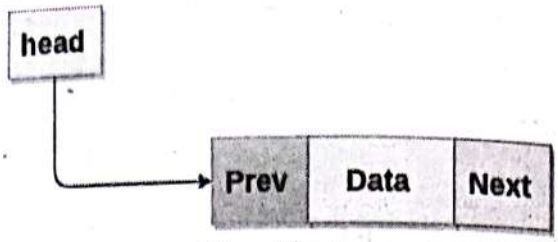



*Fig.: Node*

- A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.

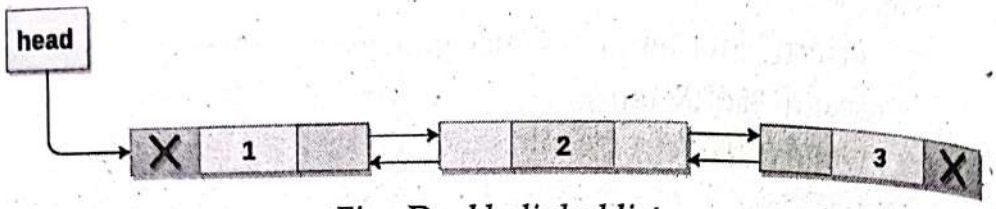



*Fig.: Doubly linked list*

In C, structure of a node in doubly linked list can be given as :

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

The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

In a singly linked list, we could traverse only in one direction, because each node contains the address of the next node and it doesn't have any record of its previous nodes. However, doubly linked lists overcome this limitation of singly linked lists. Due to the fact that each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

**Advantages of Doubly Linked List over the singly linked list:**

- A DLL can be traversed in both forward and backward directions.
- The delete operation in DLL is more efficient if a pointer to the node to be deleted is given.
- We can quickly insert a new node before a given node.
- In a singly linked list, to delete a node, a pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using the previous pointer.

**Disadvantages of Doubly Linked List over the singly linked list:**

- Every node of DLL Requires extra space for a previous pointer. It is possible to implement DLL with a single pointer though (See this and this).
- All operations require an extra pointer (previous) to be maintained. For example, in insertion, we need to modify previous pointers together with the next pointers. For example in the following functions for insertions at different positions, we need 1 or 2 extra steps to set the previous pointer.

**Applications of Doubly Linked List:**

- It is used by web browsers for backward and forward navigation of web pages
- LRU (Least Recently Used) / MRU (Most Recently Used) Caches are constructed using Doubly Linked Lists.
- Used by various applications to maintain undo and redo functionalities.
- In Operating Systems, a doubly linked list is maintained by thread scheduler to keep track of processes that are being executed at that time.

**Operations on Doubly Linked List**

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

1. Insertion
2. Deletion
3. Display

**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

**Inserting At Beginning of the list**

We can use the following steps to insert a new node at the 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.**

### Inserting At End of the list

We can use the following steps to insert a new node at the 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 initialise with **head.**

**Step 5:** Keep moving the **temp** to its next node until it reaches 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.**

### 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 double linked list...

**Step 1:** Create a **newNode** with a given value.

**Step 2:** Check whether list is **Empty (head == NULL)**

**Step 3:** If it is **Empty** then, assign **NULL** to both **newNode → previous** & **newNode → next** and set **newNode** to **head.**

**Step 4:** If it is **not Empty** then, define two node pointers **temp1** & **temp2** and initialise **temp1** with **head.**

**Step 5:** Keep moving the **temp1** to its next node until it reaches 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 is not possible!!!'** and terminate the function. Otherwise move the **temp1** to the 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.**

## 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

### Deleting from Beginning of the list

We can use the following steps to delete a node from the 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 terminates the function.

**Step 3:** If it is not Empty then, define a Node pointer **'temp'** and initialise 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.**

### Deleting from End of the list

We can use the following steps to delete a node from the 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 terminates the function.

**Step 3:** If it is not Empty then, define a Node pointer **'temp'** and initialise 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.**

### 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 terminates the function.

**Step 3:** If it is not Empty, then define a Node pointer **'temp'** and initialise with **head.**

**Step 4:** Keep moving the **temp** until it reaches 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 is 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 or not

**Step 7:** If the 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 the 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** or 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)).**

### Displaying a Double Linked List

We can use the following steps to display the elements of a double 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 initialise 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 an arrow pointing to **NULL** (**temp → data → NULL**).

## Implementation of Doubly Linked List using C Programming

```
#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;
 clrscr();
 while(1)
 {
  printf("\n********** MENU **********\n");
  printf("1. Insert\n2. Delete\n3. Display\n4. Exit\nEnter your
  choice: ");
  scanf("%d",&choice1);
  switch()
  {
   case 1: printf("Enter the value to be inserted: ");
           scanf("%d",&value);
       while(1)
       {
  printf("\nSelect from the following Inserting options\n");
  printf("1. At Beginning\n2. At End\n3. After a Node\n4.
  Cancel\nEnter your choice: ");
           scanf("%d",&choice2);
```

```
        switch(choice2)
        {
          case 1: insertAtBeginning(value);
                  break;
          case 2: insertAtEnd(value);
                  break;
          case 3: printf("Enter the location after which you
want to insert: ");
                  scanf("%d",&location);
                  insertAfter(value,location);
                  break;
          case 4: goto EndSwitch;
          default: printf("\nPlease select correct Inserting
option!!!\n");
        }
      }
 case 2: while(1)
      {
printf("\nSelect from the following Deleting options\n");
printf("1. At Beginning\n2. At End\n3. Specific Node\n4.
Cancel\nEnter your choice: ");
        scanf("%d",&choice2);
        switch(choice2)
        {
          case 1: deleteBeginning();
                  break;
          case 2: deleteEnd();
                  break;
          case 3: printf("Enter the Node value to be deleted: ");
                  scanf("%d",&location);
                  deleteSpecic(location);
                  break;
          case 4: goto EndSwitch;
default: 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;
  newNode -> previous = NULL;
  if(head == NULL)
  {
    newNode -> next = NULL;
    head = newNode;
  }
  else
  {
    newNode -> next = head;
    head = newNode;
  }
  printf("\nInsertion success!!!");
}
void insertAtEnd(int value)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode -> data = value;
  newNode -> next = NULL;
  if(head == NULL)
  {
    newNode -> previous = NULL;
    head = newNode;
  }
  else
```

```
  {
    struct Node *temp = head;
    while(temp -> next != NULL)
      temp = temp -> next;
    temp -> next = newNode;
    newNode -> previous = temp;
  }
  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)
  {
    newNode -> previous = newNode -> next = NULL;
    head = newNode;
  }
  else
  {
    struct Node *temp1 = head, temp2;
    while(temp1 -> data != location)
    {
      if(temp1 -> next == NULL)
      {
        printf("Given node is not found in the list!!!");
        goto EndFunction;
      }
      else
      {
        temp1 = temp1 -> next;
      }
    }
    temp2 = temp1 -> next;
    temp1 -> next = newNode;
    newNode -> previous = temp1;
```

```
        newNode -> next = temp2;
        temp2 -> previous = newNode;
        printf("\nInsertion success!!!");
      }
      EndFunction:
    }
    void deleteBeginning()
    {
      if(head == NULL)
        printf("List is Empty!!! Deletion not possible!!!");
      else
      {
        struct Node *temp = head;
        if(temp -> previous == temp -> next)
        {
          head = NULL;
          free(temp);
        }
        else{
          head = temp -> next;
          head -> previous = NULL;
          free(temp);
        }
        printf("\nDeletion success!!!");
      }
    }
    void deleteEnd()
    {
      if(head == NULL)
        printf("List is Empty!!! Deletion not possible!!!");
      else
      {
        struct Node *temp = head;
        if(temp -> previous == temp -> next)
        {
          head = NULL;
          free(temp);
```

```
        free(temp);
      }
      printf("\nDeletion success!!!");
    }
    FuctionEnd:
  }
  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);
    }
  }
```

**Searching for a specific node in Doubly Linked List**

We just need to traverse the list in order to search for a specific element in the list. Perform following operations in order to search a specific operation.

- Copy head pointer into a temporary pointer variable ptr.
  ptr = head
- declare a local variable I and assign it to 0.
  i=0
- Traverse the list until the pointer ptr becomes null. Keep shifting pointer to its next and increasing i by +1.
- Compare each element of the list with the item which is to be searched.
- If the item matched with any node value then the location of that value I will be returned from the function else NULL is returned.

**Algorithm**

**Step 1:** IF HEAD == NULL
WRITE "UNDERFLOW"
GOTO STEP 8
[END OF IF]

**Step 2:** Set PTR = HEAD

**Step 3:** Set i = 0

**Step 4:** Repeat step 5 to 7 while PTR != NULL

**Step 5:** IF PTR → data = item
return i
[END OF IF]

**Step 6:** i = i + 1

**Step 7:** PTR = PTR → next

**Step 8:** Exit

**C Function**

```
#include<stdio.h>
#include<stdlib.h>
void create(int);
void search();
struct node
{
  int data;
  struct node *next;
  struct node *prev;
};
struct node *head;
void main ()
{
  int choice,item,loc;
  do
  {
printf("\n1.Create\n2.Search\n3.Exit\n4.Enter your choice?");
scanf("%d",&choice);
switch(choice)
{
      case 1:
```

```
        printf("\nEnter the item\n");
        scanf("%d",&item);
        create(item);
        break;
        case 2:
        search();
        case 3:
        exit(0);
        break;
        default:
        printf("\nPlease enter valid choice\n");
      }

   }while(choice != 3);
}
void create(int item)
{

   struct node *ptr = (struct node *)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
      printf("\nOVERFLOW");
   }
   else
   {

   if(head==NULL)
   {
      ptr->next = NULL;
      ptr->prev=NULL;
      ptr->data=item;
      head=ptr;
   }
   else
   {
      ptr->data=item;printf("\nPress 0 to insert more ?\n");
      ptr->prev=NULL;
```

```
    ptr->next = head;
    head->prev=ptr;
    head=ptr;
  }
  printf("\nNode Inserted\n");
}

}
void search()
{
  struct node *ptr;
  int item,i=0,flag;
  ptr = head;
  if(ptr == NULL)
  {
    printf("\nEmpty List\n");
  }
  else
  {
    printf("\nEnter item which you want to search?\n");
    scanf("%d",&item);
    while (ptr!=NULL)
    {
      if(ptr->data == item)
      {
        printf("\nitem found at location %d ",i+1);
        flag=0;
        break;
      }
      else
      {
        flag=1;
      }
      i++;
      ptr = ptr -> next;
    }
    if(flag==1)
```

```
        {
          printf("\nItem not found\n");
        }
      }
    }
```

### Traversing in doubly linked list

Traversing is the most common operation in case of each data structure. For this purpose, copy the head pointer in any of the temporary pointer ptr.

Ptr = head

then, traverse through the list by using a while loop. Keep shifting the value of the pointer variable **ptr** until we find the last node. The last node contains **null** in its next part.

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

Although, traversing means visiting each node of the list once to perform some specific operation. Here, we are printing the data associated with each node of the list.

### Algorithm

**Step 1:** IF HEAD == NULL
WRITE "UNDERFLOW"
GOTO STEP 6
[END OF IF]

**Step 2:** Set PTR = HEAD

**Step 3:** Repeat step 4 and 5 while PTR != NULL

**Step 4:** Write PTR → data

**Step 5:** PTR = PTR → next

**Step 6:** Exit

### C Function

```
#include<stdio.h>
#include<stdlib.h>
void create(int);
int traverse();
struct node
```

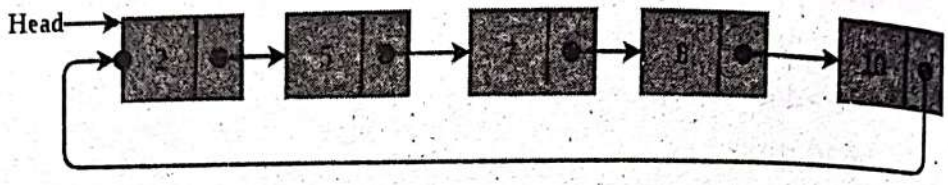
```
{
  int data;
  struct node *next;
  struct node *prev;
};
struct node *head;
void main ()
{
  int choice,item;
  do
  {
    printf("1.Append List\n2.Traverse\n3.Exit\n4.Enter your
    choice?");
    scanf("%d",&choice);
    switch(choice)
      {
      case 1:
      printf("\nEnter the item\n");
      scanf("%d",&item);
      create(item);
      break;
      case 2:
      traverse();
      break;
      case 3:
      exit(0);
      break;
      default:
      printf("\nPlease enter valid choice\n");
    }

  }while(choice != 3);
}
void create(int item)
{
  struct node *ptr = (struct node *)malloc(sizeof(struct node));
  if(ptr == NULL)
```

```
    {
      printf("\nOVERFLOW\n");
    }
    else
    {
    if(head==NULL)
    {
      ptr->next = NULL;
      ptr->prev=NULL;
      ptr->data=item;
      head=ptr;
    }
    else
    {
      ptr->data=item;printf("\nPress 0 to insert more ?\n");
      ptr->prev=NULL;
      ptr->next = head;
      head->prev=ptr;
      head=ptr;
    }
    printf("\nNode Inserted\n");
  }
  }
  int traverse()
  {
    struct node *ptr;
    if(head == NULL)
    {
      printf("\nEmpty List\n");
    }
    else
    {
      ptr = head;
      while(ptr != NULL)
      {
        printf("%d\n",ptr->data);
        ptr=ptr->next;
```

```
    }
  }
}
```

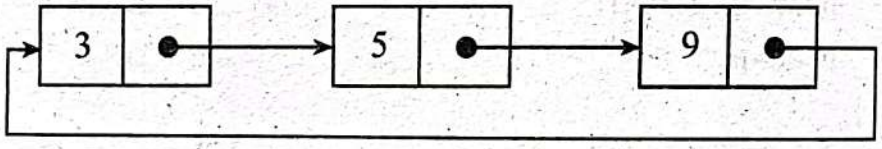
## Circular Linked List

The circular linked list is a linked list where all nodes are connected to form a circle. In a circular linked list, the first node and the last node are connected to each other which forms a circle. There is no NULL at the end.



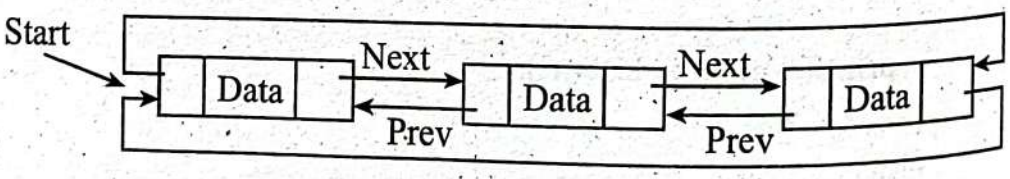

**There are generally two types of circular linked lists:**

- **Circular singly linked list:** In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We traverse the circular singly linked list until we reach the same node where we started. The circular singly linked list has no beginning or end. No null value is present in the next part of any of the nodes.




*Representation of Circular singly linked list*

- **Circular Doubly linked list:** Circular Doubly Linked List has properties of both doubly linked list and circular linked list in which two consecutive elements are linked or connected by the previous and next pointer and the last node points to the first node by the next pointer and also the first node points to the last node by the previous pointer.




*Representation of circular doubly linked list*

**Note:** We will be using the singly circular linked list to represent the working of the circular linked list.

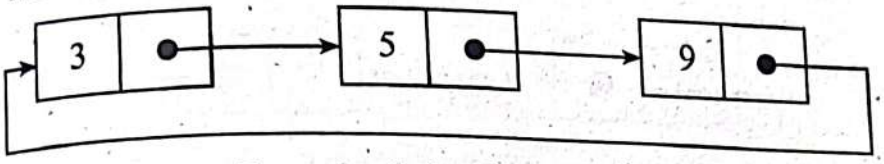
## Representation of circular linked list:

Circular linked lists are similar to single Linked Lists with the exception of connecting the last node to the first node.

Node representation of a Circular Linked List.

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

**Example of Circular singly linked list:**




*Example of circular linked list*

**The above Circular singly linked list can be represented as:**

```
Node* one = createNode(3);
Node* two = createNode(5);
Node* three = createNode(9);

// Connect nodes
one->next = two;
two->next = three;
three->next = one;
```

**Explanation:** In the above program one, two, and three are the node with values 3, 5, and 9 respectively which are connected in a circular manner as:

- **For Node One:** The Next pointer stores the address of Node two.
- **For Node Two:** The Next stores the address of Node three
- **For Node Three:** The Next points to node one.

## Operations on the circular linked list:

We can do some operations on the circular linked list similar to the singly linked list which are:

1. Insertion
2. Deletion

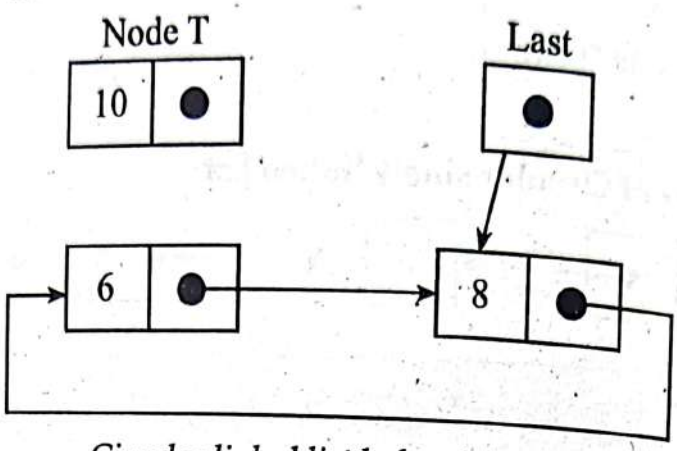
### 1. Insertion in the circular linked list:

A node can be added in three ways:

1. Insertion at the beginning of the list
2. Insertion at the end of the list
3. Insertion in between the nodes

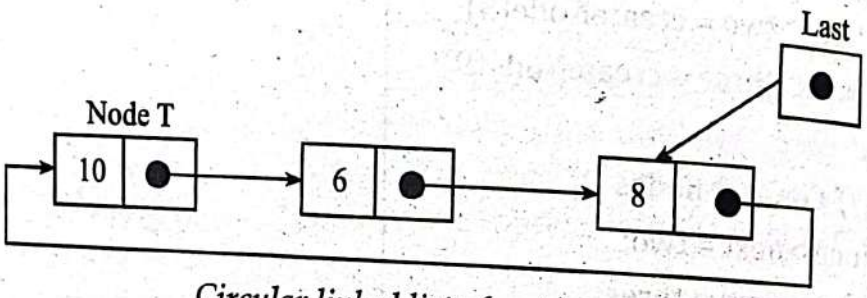
1. **Insertion at the beginning of the list:** To insert a node at the beginning of the list, follow these steps:

- Create a node, say T.
- Make T -> next = last -> next.
- last -> next = T.




*Circular linked list before insertion*

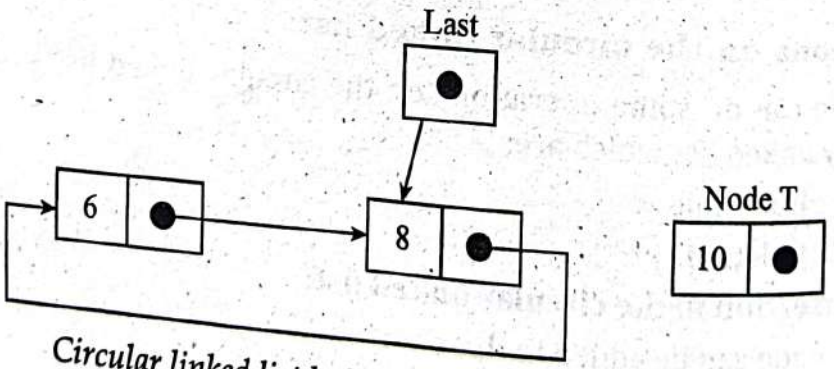
And then,




*Circular linked list after insertion*

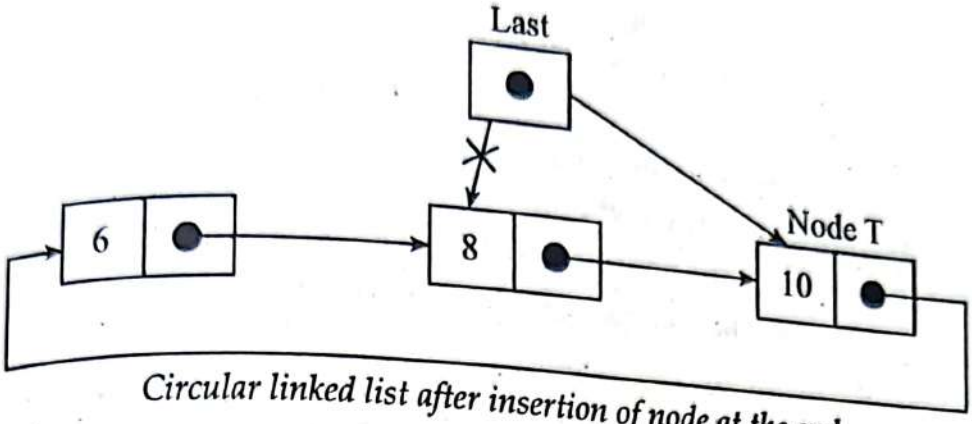
2. **Insertion at the end of the list:** To insert a node at the end of the list, follow these steps:
   - Create a node, say T.
   - Make T -> next = last -> next;
   - last -> next = T.
   - last = T.

   Before insertion,




*Circular linked list before insertion of node at the end*

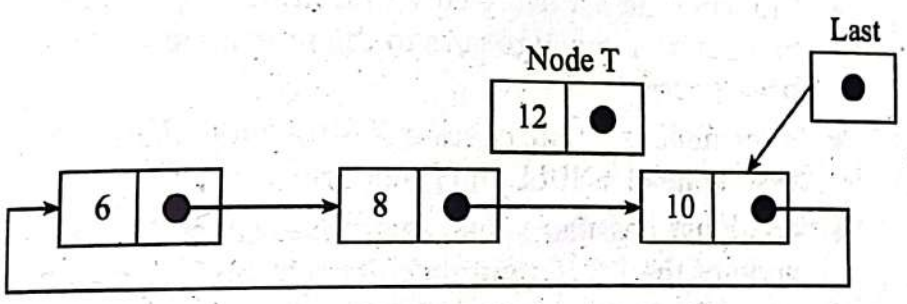
After insertion,




*Circular linked list after insertion of node at the end*

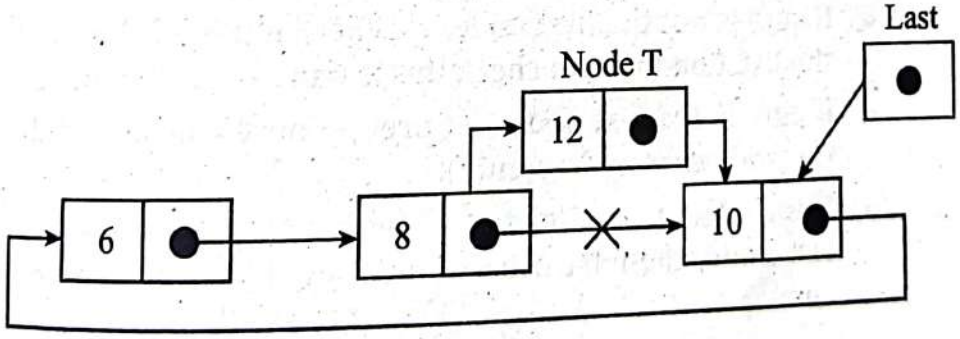
3. **Insertion in between the nodes:** To insert a node in between the two nodes, follow these steps:
   - Create a node, say T.
   - Search for the node after which T needs to be inserted, say that node is P.
   - Make T -> next = P -> next;
   - P -> next = T.

   Suppose 12 needs to be inserted after the node has the value 10,




*Circular linked list before insertion*

After searching and insertion,




*Circular linked list after insertion*

2. **Deletion in a circular linked list:**
   1. **Delete the node only if it is the only node in the circular linked list:**
      - Free the node's memory

- The last value should be NULL A node always points to another node, so NULL assignment is not necessary.

  Any node can be set as the starting point.

  Nodes are traversed quickly from the first to the last.

2. **Deletion of the last node:**
   - Locate the node before the last node (let it be temp)
   - Keep the address of the node next to the last node in temp
   - Delete the last memory
   - Put temp at the end
3. **Delete any node from the circular linked list:** We will be given a node and our task is to delete that node from the circular linked list.

   **Algorithm:**

   **Case 1:** List is empty.
   - If the list is empty we will simply return.

   **Case 2:** List is not empty
   - If the list is not empty then we define two pointers curr and prev and initialise the pointer curr with the head node.
   - Traverse the list using curr to find the node to be deleted and before moving to curr to the next node, every time set prev = curr.
   - If the node is found, check if it is the only node in the list. If yes, set head = NULL and free(curr).
   - If the list has more than one node, check if it is the first node of the list. Condition to check this( curr == head). If yes, then move prev until it reaches the last node. After prev reaches the last node, set head = head -> next and prev -> next = head. Delete curr.
   - If curr is not the first node, we check if it is the last node in the list. Condition to check this is (curr -> next == head).
   - If curr is the last node. Set prev -> next = head and delete the node curr by free(curr).
   - If the node to be deleted is neither the first node nor the last node, then set prev -> next = curr -> next and delete curr.
   - If the node is not present in the list, return head and don't do anything.

**Below is the implementation for the above approach:**

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

```
// Structure for a node
struct Node {
  int data;
  struct Node* next;
};

// Function to insert a node at the
// beginning of a Circular linked list
void push(struct Node** head_ref, int data)
{

  // Create a new node and make head
  // as next of it.
  struct Node* ptr1 = (struct Node*)malloc(sizeof(struct Node));
  ptr1->data = data;
  ptr1->next = *head_ref;

  // If linked list is not NULL then
  // set the next of last node
  if (*head_ref != NULL) {

    // Find the node before head and
    // update next to it.
    struct Node* temp = *head_ref;
    while (temp->next != *head_ref)
      temp = temp->next;
    temp->next = ptr1;
  }
  else

    // For the first node
    ptr1->next = ptr1;

  *head_ref = ptr1;
}

// Function to print nodes in a given
```

```
// circular linked list
void printList(struct Node* head)
{
  struct Node* temp = head;
  if (head != NULL) {
    do {
      printf("%d ", temp->data);
      temp = temp->next;
    } while (temp != head);
  }

  printf("\n");
}
// Function to delete a given node
// from the list
void deleteNode(struct Node** head, int key)
{
  // If linked list is empty
  if (*head == NULL)
    return;

  // If the list contains only a
  // single node
  if ((*head)->data == key && (*head)->next == *head) {
    free(*head);
    *head = NULL;
    return;
  }
  struct Node *last = *head, *d;
  // If head is to be deleted
  if ((*head)->data == key) {
  // Find the last node of the list
  while (last->next != *head)
      last = last->next;
    // Point last node to the next of
    // head i.e. the second node
    // of the list
```

```
    last->next = (*head)->next;
    free(*head);
    *head = last->next;
    return;
  }
  // Either the node to be deleted is
  // not found or the end of list
  // is not reached
  while (last->next != *head && last->next->data != key) {
    last = last->next;
  }
  // If node to be deleted was found
  if (last->next->data == key) {
    d = last->next;
    last->next = d->next;
    free(d);
  }
  else
    printf("Given node is not found in the list!!!\n");
}
// Driver code
int main()
{
  // Initialize lists as empty
  struct Node* head = NULL;
  // Created linked list will be
  // 2->5->7->8->10
  push(&head, 2);
  push(&head, 5);
  push(&head, 7);
  push(&head, 8);
  push(&head, 10);
  printf("List Before Deletion: ");
  printList(head);
  deleteNode(&head, 7);
  printf("List After Deletion: ");
  printList(head);
  return 0;
}
```

**Output**

List Before Deletion: 10 8 7 5 2

List After Deletion: 10 8 5 2

Time Complexity: O(N), Worst case occurs when the element to be deleted is the last element and we need to move through the whole list.

Auxiliary Space: O(1), As constant extra space is used.

**Advantages of Circular Linked Lists:**

- Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
- Useful for implementation of a queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use a circular linked list. We can maintain a pointer to the last inserted node and the front can always be obtained as next or last.
- Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.
- Circular Doubly Linked Lists are used for the implementation of advanced data structures like the Fibonacci Heap.
- Implementing a circular linked list can be relatively easy compared to other more complex data structures like trees or graphs.

**Disadvantages of circular linked list:**

- Compared to singly linked lists, circular lists are more complex.
- Reversing a circular list is more complicated than singly or doubly reversing a circular list.
- It is possible for the code to go into an infinite loop if it is not handled carefully.
- It is harder to find the end of the list and control the loop.
- Although circular linked lists can be efficient in certain applications, their performance can be slower than other data structures in certain cases, such as when the list needs to be sorted or searched.
- Circular linked lists don't provide direct access to individual nodes

**Applications of circular linked lists:**

- Multiplayer games use this to give each player a chance to play.

- A circular linked list can be used to organise multiple running applications on an operating system. These applications are iterated over by the OS.
- Circular linked lists can be used in resource allocation problems.
- Circular linked lists are commonly used to implement circular buffers,
- Circular linked lists can be used in simulation and gaming.

**Why circular linked lists?**

- A node always points to another node, so NULL assignment is not necessary.
- Any node can be set as the starting point.
- Nodes are traversed quickly from the first to the last.

**Stack using linked list**

**Algorithm for push() operation using linked list**

Suppose the Top is the pointer, which is pointing towards the topmost element of the stack. The top is null when the stack is empty. DATA is the data item to be pushed.

1. Input the DATA to be pushed.
2. Create a newnode
3. Newnode → DATA = DATA
4. newnode → next = Top
5. Exit.

**Algorithm for pop() operation using linked list**

Suppose Top is a pointer, which is pointing towards the topmost element of the stack. tmp is a pointer variable to hold any node's address. DATA is information on the node which is just deleted.

1. if(Top=NULL)
   Display "empty stack"
2. else
   a. tmp=top
   b. Display "The popped element element top →data"
   c. top=top→next
   d. tmp→next=NULL
   e. Free the tmp node
   f. Exit

**The following implementation shows the stack using linked list**

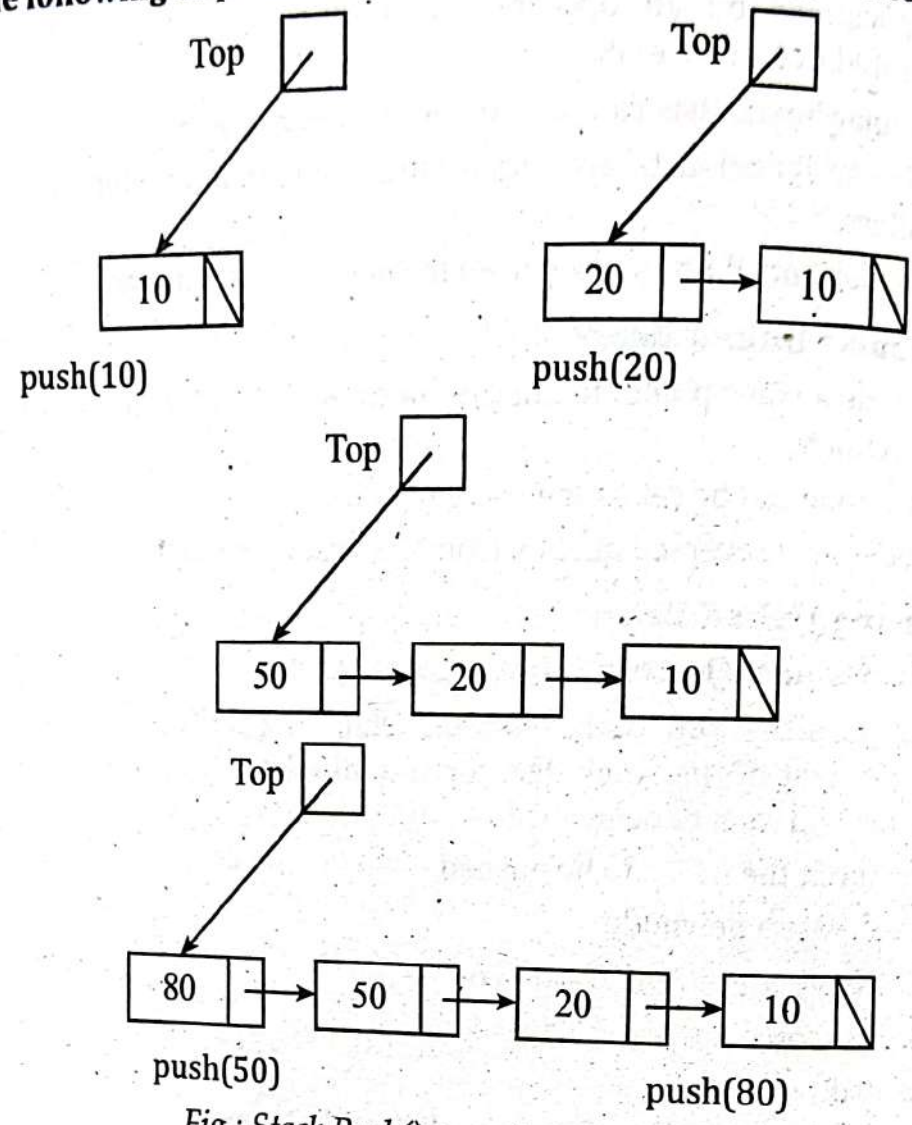



*Fig.: Stack Push() operation using linked list*

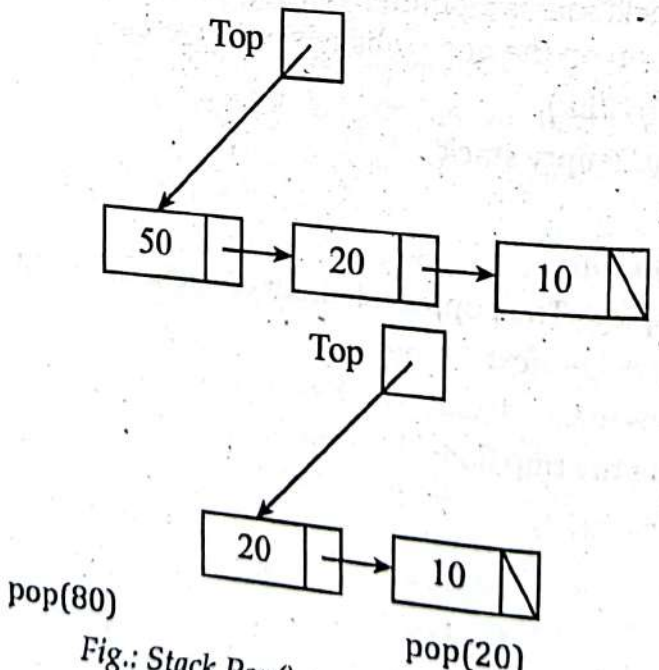



*Fig.: Stack Pop() operation using linked list*

**The following c programming implementation shows the stack using linked list.**

```
#include <stdio.h>
#include <stdlib.h>
void push();
void pop();
void display();
struct node
{
int val;
struct node *next;
};
struct node *head;

void main ()
{
  int choice=0;
  printf("\n*********Stack operations using linked
   list*********\n");
  printf("\n----------------------------------------\n");
  while(choice != 4)
  {
    printf("\n\nChose one from the below options...\n");
    printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
    printf("\n Enter your choice \n");
    scanf("%d",&choice);
    switch(choice)
    {
      case 1:
      {
        push();
        break;
      }
      case 2:
      {
        pop();
```

```
            break;
        }
        case 3:
        {
            display();
            break;
        }
        case 4:
        {
            printf("Exiting....");
            break;
        }
        default:
        {
            printf("Please Enter valid choice ");
        }
    };
}
}
void push ()
{
    int val;
    struct node *ptr = (struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("not able to push the element");
    }
    else
    {
        printf("Enter the value");
        scanf("%d",&val);
        if(head==NULL)
        {
            ptr->val = val;
            ptr -> next = NULL;
```

```
        head=ptr;
      }
      else
      {
        ptr->val = val;
        ptr->next = head;
        head=ptr;
      }
      printf("Item pushed");

    }
  }
  void pop()
  {
    int item;
    struct node *ptr;
    if (head == NULL)
    {
      printf("Underflow");
    }
    else
    {
      item = head->val;
      ptr = head;
      head = head->next;
      free(ptr);
      printf("Item popped");
    }
  }
  void display()
  {
    int i;
    struct node *ptr;
    ptr=head;
    if(ptr == NULL)
    {
```

```
    printf("Stack is empty\n");
  }
  else
  {
    printf("Printing Stack elements \n");
    while(ptr!=NULL)
    {
      printf("%d\n",ptr->val);
      ptr = ptr->next; }}}
```

**Queue using linked list**

**Algorithm for enqueue in a queue.**

Rear is the pointer in a queue where the new elements are added. Front is a pointer pointing to a queue where the elements are popped. DATA is an element to be pushed.

1. Input DATA element to be pussed.
2. Create a newnode.
3. Newnode → DATA=DATA
4. Newnode → next=NULL
5. If (Rear not equal to NULL)
   Rear → next = Newnode
6. Rear = Newnode
7. exit

**Algorithm for dequeue in a queue.**

Rear is the pointer in a queue where the new elements are added. Front is a pointer pointing to a queue where the elements are popped. DATA is an element to be pushed.

1. If (Front is equal to NULL)
   Display "empty queue"
2. Else
   a. Display "The popped element is Front→DATA"
   b. if(front is not equal to rear)
      font=front→next
   c. Else
   d. front=Null
3. exit/stop

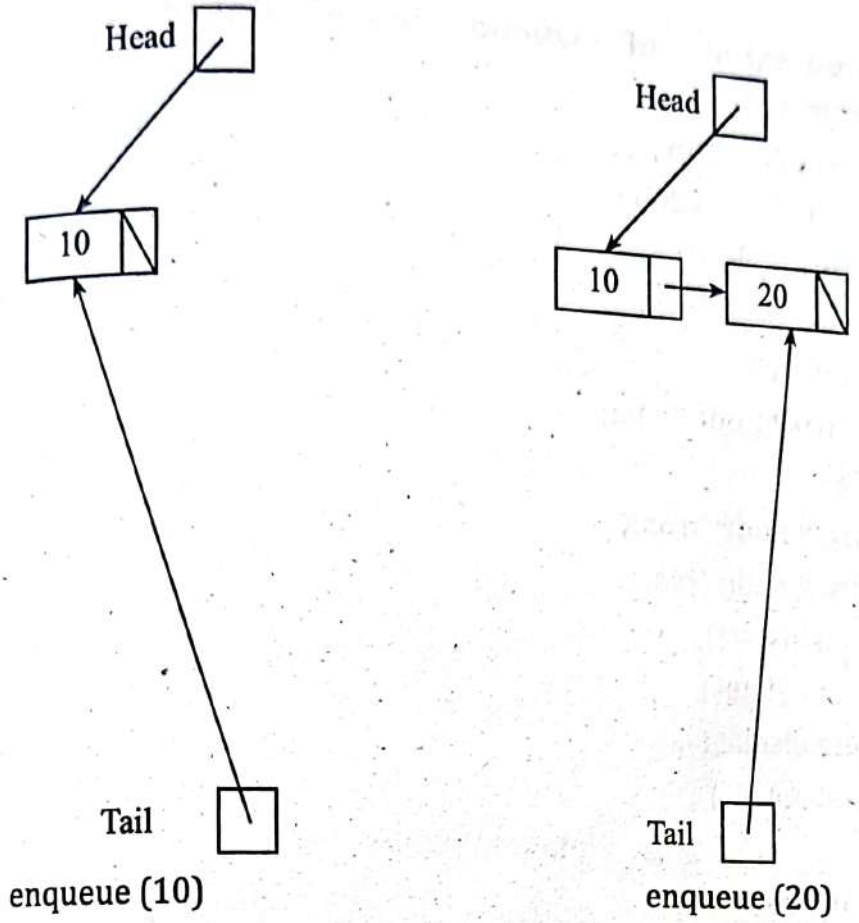



*Fig.: Queue enqueue() operation using linked list*


*Fig.: Queue dequeue() operation using linked list*

**Implementation of Queue using linked list in c programming**

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
  int data;
  struct node *next;
};
struct node *front;
struct node *rear;
void insert();
void delete();
void display();
void main ()
{
  int choice;
  while(choice != 4)
  {
    printf("\n**********Main
    Menu**********\n");

    printf("\n=========================================
    ==========\n");
    printf("\n1.insert an element\n2.Delete an
    element\n3.Display the queue\n4.Exit\n");
    printf("\nEnter your choice ?");
    scanf("%d",& choice);
    switch(choice)
    {
      case 1:
      insert();
      break;
      case 2:
      delete();
      break;
      case 3:
```

```
        display();
        break;
        case 4:
        exit(0);
        break;
        default:
        printf("\nEnter valid choice??\n");
      }
    }
  }
void insert()
{
  struct node *ptr;
  int item;

  ptr = (struct node *) malloc (sizeof(struct node));
  if(ptr == NULL)
  {
    printf("\nOVERFLOW\n");
    return;
  }
  else
  {
    printf("\nEnter value?\n");
    scanf("%d",&item);
    ptr -> data = item;
    if(front == NULL)
    {
      front = ptr;
      rear = ptr;
      front -> next = NULL;
      rear -> next = NULL;
    }
    else
    {
      rear -> next = ptr;
      rear = ptr;
```

```
            rear->next = NULL;
        }
      }
    }
}
void delete ()
{
   struct node *ptr;
   if(front == NULL)
   {
      printf("\nUNDERFLOW\n");
      return;
   }
    else
    {
       ptr = front;
       front = front -> next;
       free(ptr);
    }
}
   void display()
   {
      struct node *ptr;
      ptr = front;
      if(front == NULL)
      {
         printf("\nEmpty queue\n");
      }
       else
       {  printf("\nprinting values .....\n");
          while(ptr != NULL)
          {
             printf("\n%d\n",ptr -> data);
             ptr = ptr -> next;
          }
         }
          }
```

## Polynomial using linked list

- Different operations such as addition, subtraction, division and Multiplication of polynomials can be performed using a linked list.
- Following example shows the polynomial addition using a linked list. In the linked representation of polynomials, each term is considered as a node. And such a node contains three fields, coefficient field, exponent field and link field.

**Logical representation is,**

```
struct Polynode
{
int coeff;
int expo;
Struct Polynode *next;
}
```

consider two polynomials f(n) and g(n). It can be represented. wing linked list as follows:

$f(x) = ax^3 + bx + c$

$g(x) = mx^4 + x^3n + ax^2 + px + q$

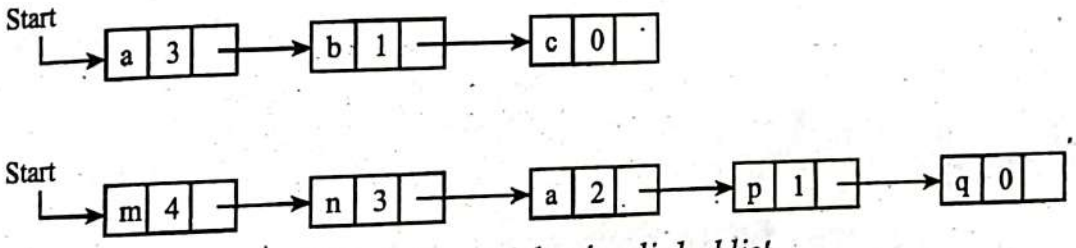



*Fig.: Polynomial using linked list*

The addition of polynomial can be represented as:

$h(x)=f(x)+g(x) = mx^4+(a+n)x^3 + ax^2 + (b+p)x + (c+q)$

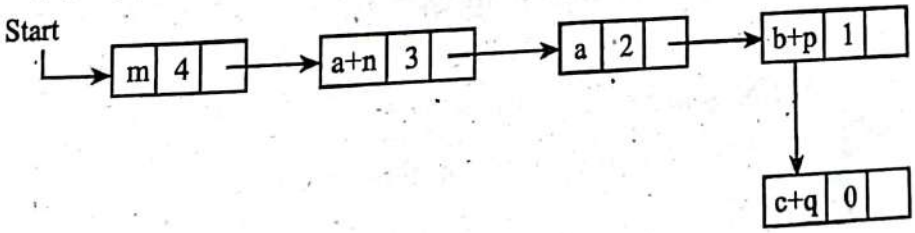



*Fig.: Addition of two polynomials using linked list*

## C++ program to add two polynomials using linked list

```
#include <iostream.h>
using namespace std;
int max(int m, int n) { return (m > n)? m: n; }
int *add(int A[], int B[], int m, int n)
```

```
{
int size = max(m, n);
int *sum = new int[size];
for (int i = 0; i<m; i++)
  sum[i] = A[i];
for (int i=0; i<n; i++)
  sum[i] += B[i];
return sum;
}
void printPoly(int poly[], int n)
{
  for (int i=0; i<n; i++)
  {
  cout << poly[i];
  if (i != 0)
    cout << "x^" << i ;
  if (i != n-1)
  cout << " + ";
  }
}
int main()
{
  int A[] = { 5, 0, 10, 6 };
  int B[] = { 1, 2, 4 };
  int m = sizeof(A)/sizeof(A[0]);
  int n = sizeof(B)/sizeof(B[0]);
  cout << "First polynomial is \n";
  printPoly(A, m);
  cout << "\n Second polynomial is \n";
  printPoly(B, n);
  int *sum = add(A, B, m, n);
  int size = max(m, n);
  cout << "\n Sum of polynomial is \n";
  printPoly(sum, size);
  return 0;
}
```

## Differences between Array, Stack, Queue, Linked list

**Array:**

- An array is a collection of elements, each identified by an index or a key.
- Elements are stored in contiguous memory locations.
- The size of the array is fixed during initialization.
- Random access to elements is efficient because of constant-time indexing.
- Insertion and deletion operations may be less efficient, especially in the middle, as elements may need to be shifted.

**Stack:**

- A stack is a Last-In-First-Out (LIFO) data structure, meaning the last element added is the first one to be removed.
- Operations are performed at one end, known as the top of the stack.
- Common operations include push (add an element to the top) and pop (remove the top element).
- Stacks are used for tasks like function call management, expression evaluation, and backtracking algorithms.

**Queue:**

- A queue is a First-In-First-Out (FIFO) data structure, meaning the first element added is the first one to be removed.
- Operations are performed at two ends, with elements added at the rear and removed from the front.
- Common operations include enqueue (add an element to the rear) and dequeue (remove an element from the front).
- Queues are used in scenarios like task scheduling, breadth-first search, and print job management.

**Linked List:**

- A linked list is a data structure where elements are stored in nodes, and each node contains a reference to the next node in the sequence.
- Elements are not stored in contiguous memory locations.
- Linked lists can be singly linked (each node points to the next) or doubly linked (each node points to both the next and the previous).
- **Dynamic size:** Nodes can be easily added or removed, allowing for efficient insertions and deletions.
- Random access is less efficient compared to arrays because it requires traversing the list.

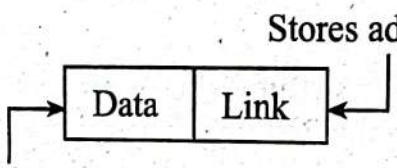
## Application of linked list

- Polynomial Manipulation representation
- Addition of long positive integers
- Representation of sparse matrices
- Addition of long positive integers
- Symbol table creation, Mailing list, Memory management, Linked allocation of files and Multiple precision arithmetic etc.

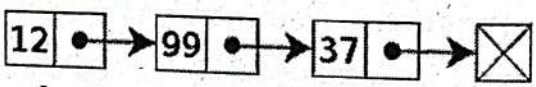
## Differences between SLL, DLL and CLL

### Singly Linked List

- Singly linked list is a sequence of elements in which every element has a link to its next element in the sequence. In any single linked list, the individual element is called a "Node". Every "Node" contains two fields, data and next. The data field is used to store the actual value of that node and the next field is used to store the address of the next node in the sequence.
- In this type of linked list, there is only one link in each node, where the link points to the next node in the list. The link of the last node has a NULL pointer.
- The graphical representation of a node in a single linked list is as follows...




The following example is a singly linked list that contains three elements 12, 99, & 37.



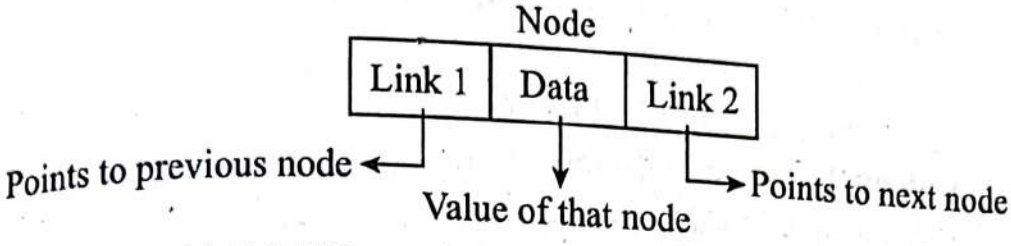

**Node Structure for the SLL is given below:**

```
// Node structure for SLL
struct Node {
  int data;
  struct Node* next;
};
```

### Doubly Linked List (DLL):

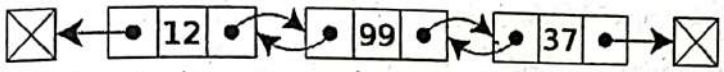
- Doubly linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.
- In a double linked list, every node has a link to its previous node and next node. So, we can traverse forward by using the next field and can traverse backward by using the previous field. Every node

in a double linked list contains three fields and they are shown in the following figure...




- 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.

**Example:**

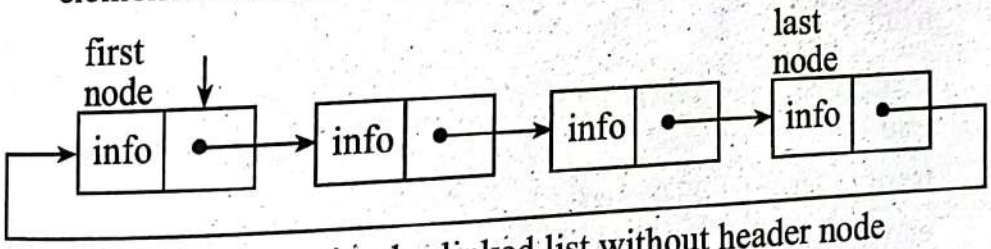



**Node Structure for the DLL is given below:**

```
// Node structure for DLL
struct Node {
  int data;
  struct Node* prev;
  struct Node* next; };
```

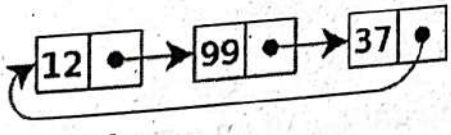
## Circular Linked List (CLL)

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




a circular linked list without header node

**Example:**




Circular list has no end.

In a circular linked list there are two methods to know if a node is the first node or not.

- Either a external pointer, *list*, points the first node or
- A *header node* is placed as the first node of the circular list.

**Node Structure for the CLL is given below:**

```
// Node structure for CLL
struct Node {   int data;   struct Node* next;  };
```

**1. Write algorithms to enqueue and dequeue in a circular queue.** [2021 Fall]

⇒ **Enqueue Operation in Circular Queue:**

**Algorithm Enqueue(queue, item)**

```
if (rear + 1) % size == front then
Print "Queue is Full"
else
if front == -1 then
front <- 0
rear <- (rear + 1) % size
queue[rear] <- item
end if
```

**Dequeue Operation in Circular Queue:**

```
Algorithm Dequeue(queue)
if front == -1 then
Print "Queue is Empty"
return -1
else
item <- queue[front]
if front == rear then
front <- -1
rear <- -1
else
front <- (front + 1) % size
end if
return item
end if
```

**2. What is the advantage of an array over a linked list and what is the advantage of a linked list over an array? Implement a simple singly linked list with three nodes containing data 11, 22, and 33 using C or C++ code.**

⇒ **Arrays vs. Linked Lists**

**Advantages of Arrays:**

- **Direct access:** Arrays provide O(1) time complexity for accessing elements using an index.

- **Memory locality**: Arrays have better cache performance due to contiguous memory allocation.

**Advantages of Linked Lists:**

- **Dynamic size**: Linked lists can grow and shrink in size dynamically.
- **Efficient insertions/deletions**: Insertions and deletions are O(1) operations if the position is known, unlike arrays that require shifting elements.

**Singly Linked List Implementation in C++:**

```
#include <iostream>
using namespace std;

struct Node {
  int data;
  Node* next;
};

int main() {
  // Create nodes
  Node* first = new Node();
  Node* second = new Node();
  Node* third = new Node();

  // Assign data
  first->data = 11;
  second->data = 22;
  third->data = 33;

  // Link nodes
  first->next = second;
  second->next = third;
  third->next = nullptr;

  // Print linked list
  Node* temp = first;
  while(temp != nullptr) {
    cout << temp->data << " ";
```

```
        temp = temp->next;
      }
      return 0;
    }
```

**3. Write algorithms for push and pop operations on a stack using linked list implementation.**

⇒ **Stack Operations Using Linked List**

**Push Operation:**

```
Algorithm Push(stack, item)
  newNode ← new Node(item)
  newNode.next ← stack.top
  stack.top ← newNode
```

**Pop Operation:**

```
Algorithm Pop(stack)
  if (stack.top == null)
    throw StackUnderflowException
  else
    item ← stack.top.data
    temp ← stack.top
    stack.top ← stack.top.next
    free(temp)
  return item
```

**4. Write algorithms to insert and delete a node at the end of a singly linked list.**

⇒ **Singly Linked List Insert/Delete at End**

**Insert at End:**

```
Algorithm InsertAtEnd(list, item)
  newNode ← new Node(item)
  if (list.head == null)
    list.head ← newNode
  else
    temp ← list.head
    while (temp.next != null)
      temp ← temp.next
    temp.next ← newNode
```

**Delete at End:**

```
Algorithm DeleteAtEnd(list)
  if (list.head == null)
    throw ListUnderflowException
  else if (list.head.next == null)
    free(list.head)
    list.head ← null
  else
    temp ← list.head
    while (temp.next.next != null)
      temp ← temp.next
    free(temp.next)
    temp.next ← null
```

---

**5. What is a double-ended queue? Explain a real-world scenario in which the double-ended queue can be applied.** [2023 Spring]

⇒ **Double-Ended Queue (Deque):** A double-ended queue (deque) is a data structure that allows insertion and deletion of elements at both the front and the rear ends. It can function both as a queue and as a stack, providing more flexibility in data management.

**Real-World Scenario:** A real-world scenario for a deque is a task scheduler in a desktop operating system where tasks need to be managed dynamically. For example, a user might add tasks to the front of the deque for urgent tasks and to the rear for less urgent tasks. Similarly, completed tasks can be removed from both ends, depending on their priority.

---

**6. Write an algorithm to delete the last node in a singly circular linked list and also implement it using C or C++ code.**

⇒ **Algorithm:**

1. If the list is empty, return.
2. If there is only one node, delete it and set the head to null.
3. Traverse the list to find the second last node.
4. Set the next pointer of the second last node to head.
5. Delete the last node.

**C++ Code Implementation:**

```
#include <iostream>
using namespace std;
struct Node {
```

```
    int data;
    Node* next;
};

void deleteLastNode(Node*& head) {
    if (head == nullptr) return; // Empty list

    if (head->next == head) {
        delete head;
        head = nullptr;
        return;
    }

    Node* temp = head;
    while (temp->next->next != head) {
        temp = temp->next;
    }

    Node* lastNode = temp->next;
    temp->next = head;
    delete lastNode;
}

void printList(Node* head) {
    if (head == nullptr) return;
    Node* temp = head;
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
}

int main() {
    Node* head = new Node{1, nullptr};
    head->next = head;
    head->next = new Node{2, head};
```

```
    head->next->next = new Node{3, head};

    cout << "Original List: ";
    printList(head);

    deleteLastNode(head);

    cout << "After Deleting Last Node: ";
    printList(head);

    return 0;
}
```

**7. What are the advantages and disadvantages of linked list implementation over array implementation? Explain the linked implementation of a stack.** [2023 Spring 4 (a)]

⇒ **Advantages of Linked List Over Array:**

1. **Dynamic Size:** Linked lists can grow and shrink dynamically without the need for resizing.
2. **Efficient Insertions/Deletions:** Inserting or deleting elements does not require shifting elements, making these operations more efficient.

**Disadvantages of Linked List Over Array:**

1. **Memory Overhead:** Each element requires additional memory for the pointer.
2. **Sequential Access:** Accessing an element requires traversal from the head, making random access inefficient.

**Linked List Implementation of Stack:** In a linked list implementation of a stack, each node contains the data and a pointer to the next node. The top pointer points to the top of the stack. Push and pop operations are performed by updating the top pointer.

**Algorithm for Push:**

1. Create a new node.
2. Set the node's data to the value to be pushed.
3. Set the node's next pointer to the current top.
4. Update top to point to the new node.

**Algorithm for Pop:**

1. If the stack is empty, return an error.

2. Store the data from the top node.
3. Update top to point to the next node.
4. Delete the old top node.
5. Return the stored data.

**C++ Code Implementation:**

cpp

Copy code

```
#include <iostream>
using namespace std;

struct Node {
  int data;
  Node* next;
};

class Stack {
private:
  Node* top;

public:
  Stack() : top(nullptr) {}

  void push(int value) {
    Node* newNode = new Node{value, top};
    top = newNode;
  }

  int pop() {
    if (top == nullptr) {
      cout << "Stack is empty" << endl;
      return -1;
    }
    int value = top->data;
    Node* temp = top;
    top = top->next;
    delete temp;
    return value;
```

```
    }

    bool isEmpty() {
      return top == nullptr;
    }
};

int main() {
    Stack stack;
    stack.push(10);
    stack.push(20);
    stack.push(30);

    cout << "Popped: " << stack.pop() << endl;
    cout << "Popped: " << stack.pop() << endl;
    cout << "Popped: " << stack.pop() << endl;

    return 0;
}
```

**8. Explain the application area of queue. What will happen if front is equal to rear in a linear queue? Demonstrate the concept of enqueue, dequeue, and traverse operations in a linear queue with a suitable example and supporting algorithm.**

⇒ **Application Area of Queue**

- **CPU Scheduling**: Manages tasks waiting for CPU time.
- **Print Queue**: Manages print jobs in a printer.
- **Breadth-First Search (BFS)**: Uses queues for level-order traversal of graphs.

**Behavior of Front = Rear in Linear Queue:**

- **Condition**: When front equals rear, the queue is empty.

**Enqueue, Dequeue, and Traverse Operations:**

- **Enqueue(x):**
  1. Check if the queue is full.
  2. If not, increment rear and insert element x at the rear position.
- **Dequeue():**
  1. Check if the queue is empty.

2. If not, retrieve the element at the front and increment front.

- **Traverse:**
  1. Start from front and print each element until rear.

**Example:**

- Initial Queue: [ ], front = 0, rear = -1
- Enqueue 1: [1], front = 0, rear = 0
- Enqueue 2: [1, 2], front = 0, rear = 1
- Dequeue: [1, 2], front = 1, rear = 1 (element 1 removed)
- Traverse: Output elements from index 1 to 1 -> [2]

---

**9. Differentiate between singly circular linked list and singly linked list. Implement queue using a linked list.**

⇒ **Difference Between Singly Circular Linked List and Singly Linked List**

- **Singly Linked List:**
  - Structure: Nodes have a single pointer to the next node.
  - Termination: Last node points to NULL.
- **Singly Circular Linked List:**
  - Structure: Nodes have a single pointer to the next node.
  - Termination: Last node points to the first node, forming a circle.

**Queue Implementation Using Linked List:**

- **Structure:**
  - Node: Contains data and a pointer to the next node.
  - Queue: Maintains pointers to front and rear nodes.

**C++ Implementation:**

```
#include <iostream>
using namespace std;

class Node {
public:
  int data;
  Node* next;
  Node(int val) : data(val), next(NULL) {}
};

class Queue {
```

```
private:
  Node *front, *rear;
public:
  Queue() : front(NULL), rear(NULL) {}

  void enqueue(int val) {
    Node* newNode = new Node(val);
    if (rear == NULL) {
      front = rear = newNode;
      return;
    }
    rear->next = newNode;
    rear = newNode;
  }

  void dequeue() {
    if (front == NULL) return;
    Node* temp = front;
    front = front->next;
    if (front == NULL) rear = NULL;
    delete temp;
  }

  void traverse() {
    Node* temp = front;
    while (temp != NULL) {
      cout << temp->data << " ";
      temp = temp->next;
    }
    cout << endl;
  }
};

int main() {
  Queue q;
  q.enqueue(10);
  q.enqueue(20);
```

```
    q.enqueue(30);
    q.traverse();
    q.dequeue();
    q.traverse();
    return 0;
}
```

**10. How are the higher-order polynomials represented using a linked list? With necessary supporting figures, write an algorithm to insert a node in a given position of a doubly linked list.**

⇒ **Higher-Order Polynomials Using Linked List**

- **Representation**: Each term is a node with coefficients and exponents.
- **Node Structure:**
  - Data: Coefficient and exponent
  - Pointer: Next node

**Insertion Algorithm in Doubly Linked List:**

1. Create a new node.
2. Traverse the list to the desired position.
3. Adjust pointers to insert the new node.

**C++ Implementation:**

```
void insertNode(Node* head, int pos, int data) {
  Node* newNode = new Node(data);
  Node* temp = head;
  for (int i = 1; i < pos && temp != NULL; i++) {
    temp = temp->next;
  }
  newNode->next = temp->next;
  newNode->prev = temp;
  if (temp->next != NULL)
    temp->next->prev = newNode;
  temp->next = newNode;
}
```

**11. Differentiate array with linked list. Write down an algorithm to perform the insertion at the beginning of a singly linked list. Also, implement this algorithm using C++ code.** [2023 Spring]

⇒ **Arrays vs. Linked Lists and Insertion in Singly Linked List**

a. **Differences**

- **Array:**
  - Structure: Contiguous memory allocation.
  - Access: Direct access to elements using indices (O(1) time complexity).
  - Size: Fixed size.
- **Linked List:**
  - Structure: Elements (nodes) are linked using pointers.
  - Access: Sequential access to elements (O(n) time complexity for access).
  - Size: Dynamic size, can easily grow or shrink.

b. **Insertion at the Beginning of a Singly Linked List Algorithm**

1. Create a new node with the given value.
2. Set the new node's next pointer to the current head of the list.
3. Update the head to be the new node.

c. **C++ Implementation**

```
#include <iostream>

struct Node {
  int data;
  Node* next;
};

void insertAtBeginning(Node*& head, int value) {
  Node* newNode = new Node();
  newNode->data = value;
  newNode->next = head;
  head = newNode;
}

void printList(Node* head) {
  Node* current = head;
  while (current != nullptr) {
    std::cout << current->data << " ";
    current = current->next;
```

```
    }
    std::cout << std::endl;
}

int main() {
    Node* head = nullptr;  // Initialize the list as empty

    insertAtBeginning(head, 10);
    insertAtBeginning(head, 20);
    insertAtBeginning(head, 30);

    printList(head);  // Output: 30 20 10

    return 0;
}
```

**12. What are the advantages of doubly linked list over singly linked list? Explain the linked implementation of queue** [2022 Fall]

⇒ **Doubly Linked List vs. Singly Linked List and Linked Implementation of Queue**

**a. Advantages of Doubly Linked List**

- **Bidirectional Traversal:** Can be traversed in both forward and backward directions.
- **Easy Deletion:** Easier to delete a given node without needing a reference to the previous node.
- **Insertion:** Easier insertion before a given node.

**b. Linked Implementation of Queue**

A queue follows the First In, First Out (FIFO) principle. Using a linked list, you can efficiently implement a queue where:

- Enqueue (Insert at the end): Append to the tail.
- Dequeue (Remove from the front): Remove from the head.

```
#include <iostream>

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

```
class Queue {
private:
  Node* front;
  Node* rear;

public:
  Queue() {
    front = nullptr;
    rear = nullptr;
  }

  void enqueue(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = nullptr;
    if (rear == nullptr) {
      front = rear = newNode;
      return;
    }
    rear->next = newNode;
    rear = newNode;
  }

  void dequeue() {
    if (front == nullptr) {
      std::cout << "Queue is empty!" << std::endl;
      return;
    }
    Node* temp = front;
    front = front->next;
    if (front == nullptr) {
      rear = nullptr;
    }
    delete temp;
  }

  void display() {
```

```
    Node* current = front;
    while (current != nullptr) {
      std::cout << current->data << " ";
      current = current->next;
    }
    std::cout << std::endl;
  }
};

int main() {
  Queue q;
  q.enqueue(10);
  q.enqueue(20);
  q.enqueue(30);
  q.display(); // Output: 10 20 30

  q.dequeue();
  q.display(); // Output: 20 30

  return 0;
}
```

**13. How does a circular queue solve the problem of a linear queue? Implement the enqueue and dequeue operations in a circular queue using C or C++ code.** [2022 Fall]

⇒ **Circular Queue Implementation**

**a. Solving the Problem of Linear Queue**

Efficient Use of Space: Circular queues reuse the vacant space created by dequeuing, thus solving the issue of unutilized space in linear queues.

**b. Enqueue and Dequeue in Circular Queue (C++)**

```
#include <iostream>
#define SIZE 5

class CircularQueue {
  int front, rear;
  int items[SIZE];
public:
```

```
CircularQueue() {
  front = -1;
  rear = -1;
}
bool isFull() {
  return (front == 0 && rear == SIZE - 1) || (front == rear +
  1);
}
bool isEmpty() {
  return front == -1;
}
void enqueue(int element) {
  if (isFull()) {
    std::cout << "Queue is full\n";
    return;
  }
  if (front == -1) front = 0;
  rear = (rear + 1) % SIZE;
  items[rear] = element;
  std::cout << "Inserted " << element << std::endl;
}
int dequeue() {
  if (isEmpty()) {
    std::cout << "Queue is empty\n";
    return -1;
  }
  int element = items[front];
  if (front == rear) {
    front = -1;
    rear = -1;
  } else {
    front = (front + 1) % SIZE;
  }
  return element;
}
void display() {
  if (isEmpty()) {
```

```
            std::cout << "Queue is empty\n";
            return;
        }
        int i;
        for (i = front; i != rear; i = (i + 1) % SIZE) {
            std::cout << items[i] << " ";
        }
        std::cout << items[i] << std::endl;
    }
};

int main() {
    CircularQueue q;
    q.enqueue(1);
    q.enqueue(2);
    q.enqueue(3);
    q.enqueue(4);
    q.enqueue(5);
    q.display();
    q.dequeue();
    q.display();
    q.enqueue(6);
    q.display();
    return 0;
}
```

**14. In which condition do you use linked list implementation of a queue? Explain with an example.** [2022 Fall]

⇒ **When to Use Linked List Implementation**

**Dynamic Size:** When the size of the queue is not known in advance or can change frequently, linked lists are preferred as they can dynamically grow or shrink.

**Example**

```
#include <iostream>
using namespace std;

struct Node {
    int data;
```

```
    Node* next;
};

class Queue {
    Node* front;
    Node* rear;
public:
    Queue() {
        front = rear = nullptr;
    }
    void enqueue(int value) {
        Node* temp = new Node();
        temp->data = value;
        temp->next = nullptr;
        if (rear == nullptr) {
            front = rear = temp;
        } else {
            rear->next = temp;
            rear = temp;
        }
        cout << value << " enqueued to queue\n";
    }
    void dequeue() {
        if (front == nullptr) {
            cout << "Queue is empty\n";
            return;
        }
        Node* temp = front;
        front = front->next;
        if (front == nullptr) {
            rear = nullptr;
        }
        delete temp;
    }
    void display() {
        Node* temp = front;
        while (temp != nullptr) {
```

```
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};

int main() {
    Queue q;
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.display();
    q.dequeue();
    q.display();
    return 0;
}
```

**15. Write an algorithm to insert a node at the beginning of a singly linked list. Also, illustrate with an example.** [2022 Fall]

⇒ **a. Algorithm**

1. Create a new node.
2. Set the new node's next pointer to the current head.
3. Update the head to be the new node.

**b. Example**

Initial List: 10 -> 20 -> 30

Insert 5 at the beginning: New List: 5 -> 10 -> 20 -> 30

**c. Implementation in C++**

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
};

void insertAtBeginning(Node*& head, int value) {
```

```
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = head;
    head = newNode;
}

void printList(Node* head) {
    Node* current = head;
    while (current != nullptr) {
        cout << current->data << " ";
        current = current->next;
    }
    cout << endl;
}

int main() {
    Node* head = nullptr;
    insertAtBeginning(head, 10);
    insertAtBeginning(head, 20);
    insertAtBeginning(head, 30);
    printList(head); // Output: 30 20 10
    insertAtBeginning(head, 5);
    printList(head); // Output: 5 30 20 10
    return 0;
}
```

16. **Write down the condition to check weather the element in a linear queue is the last element or not. also mention how do you find the number of elements in any queue?** [2017 Fall]

⇒ To check whether an element in a linear queue is the last element, compare the element's position to the rear pointer:

```
if position == rear:
    # It is the last element
else:
    # It is not the last element
```

To find the number of elements in the queue, calculate the difference between the rear and front pointers and add 1 (assuming 0-based indexing):

```
number_of_elements = rear - front + 1
```