# DATA STRUCTURES

## UNIT-2

## Stacks & Queues

## using

## Linked List

Dr G.KALYANI

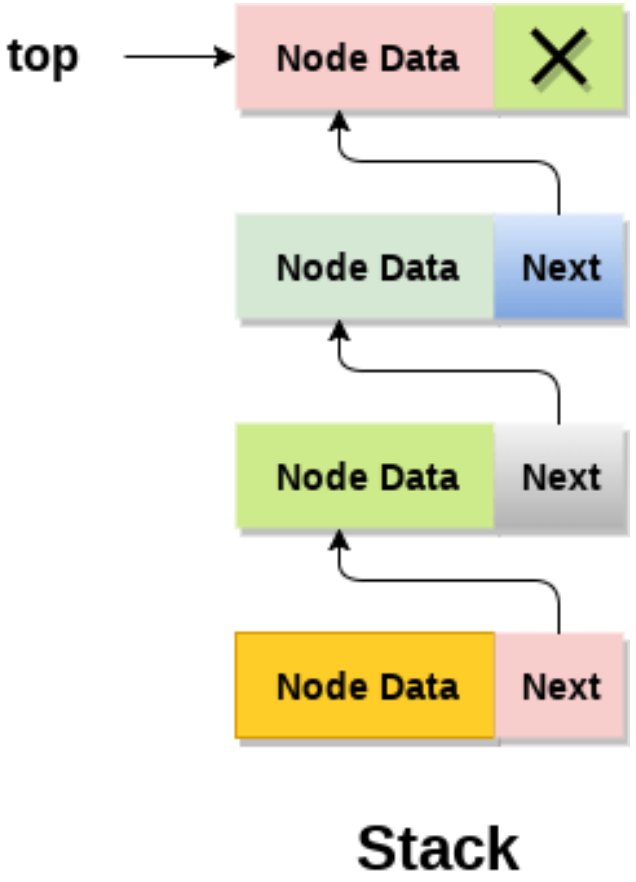
# Topics

- **Stacks using linked List**
- Queues using Linked List

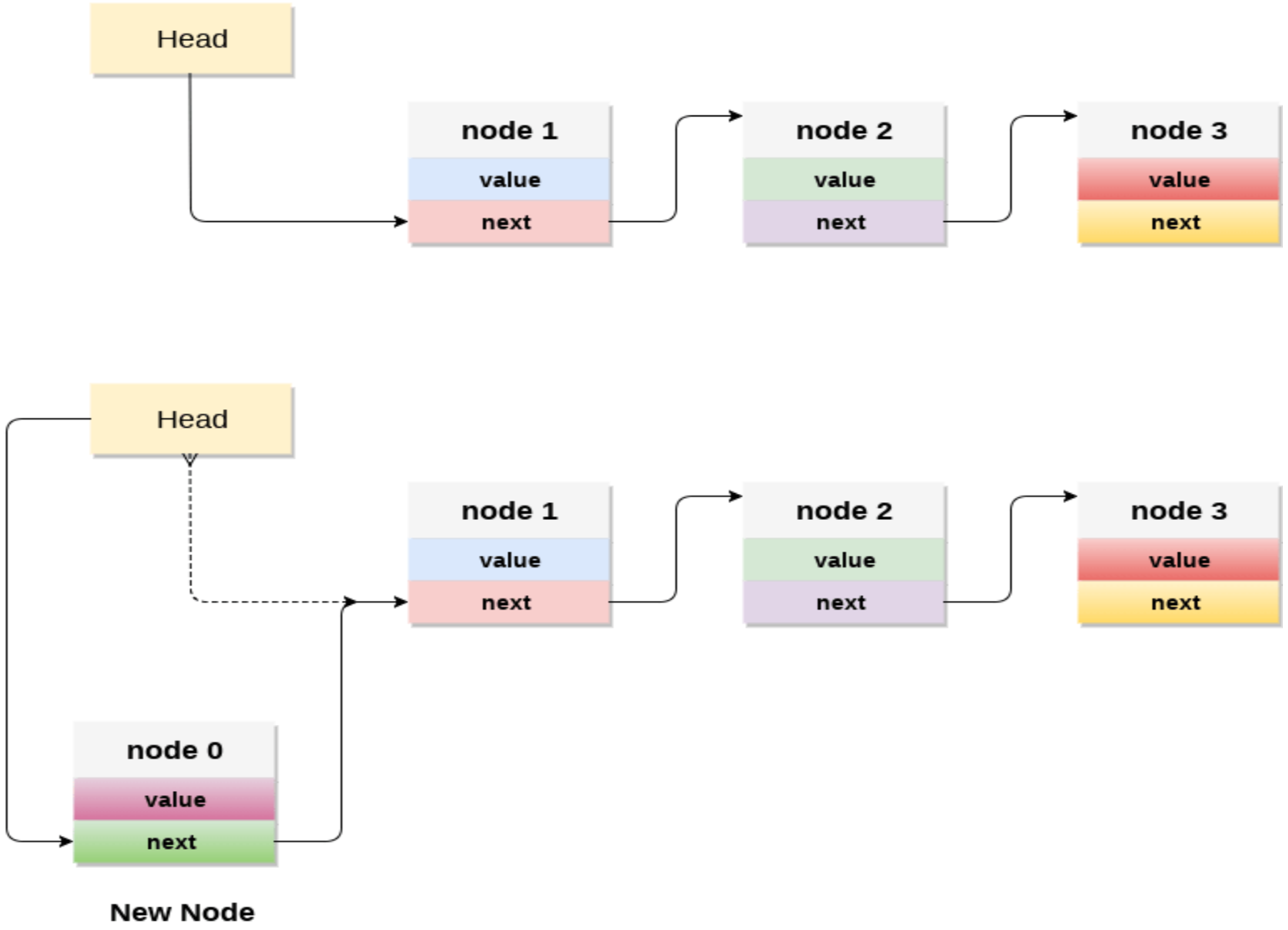
# Linked list implementation of stack

- Instead of using array, we can also use linked list to implement stack.
- Linked list allocates the memory dynamically.
- In linked list implementation of stack, the nodes are maintained non-contiguously in the memory.
- Each node contains a pointer to its immediate successor node in the stack.
- Stack is said to be overflown if the space left in the memory heap is not enough to create a node.
- The top most node in the stack always contains null in its address field.

# Linked list implementation of stack



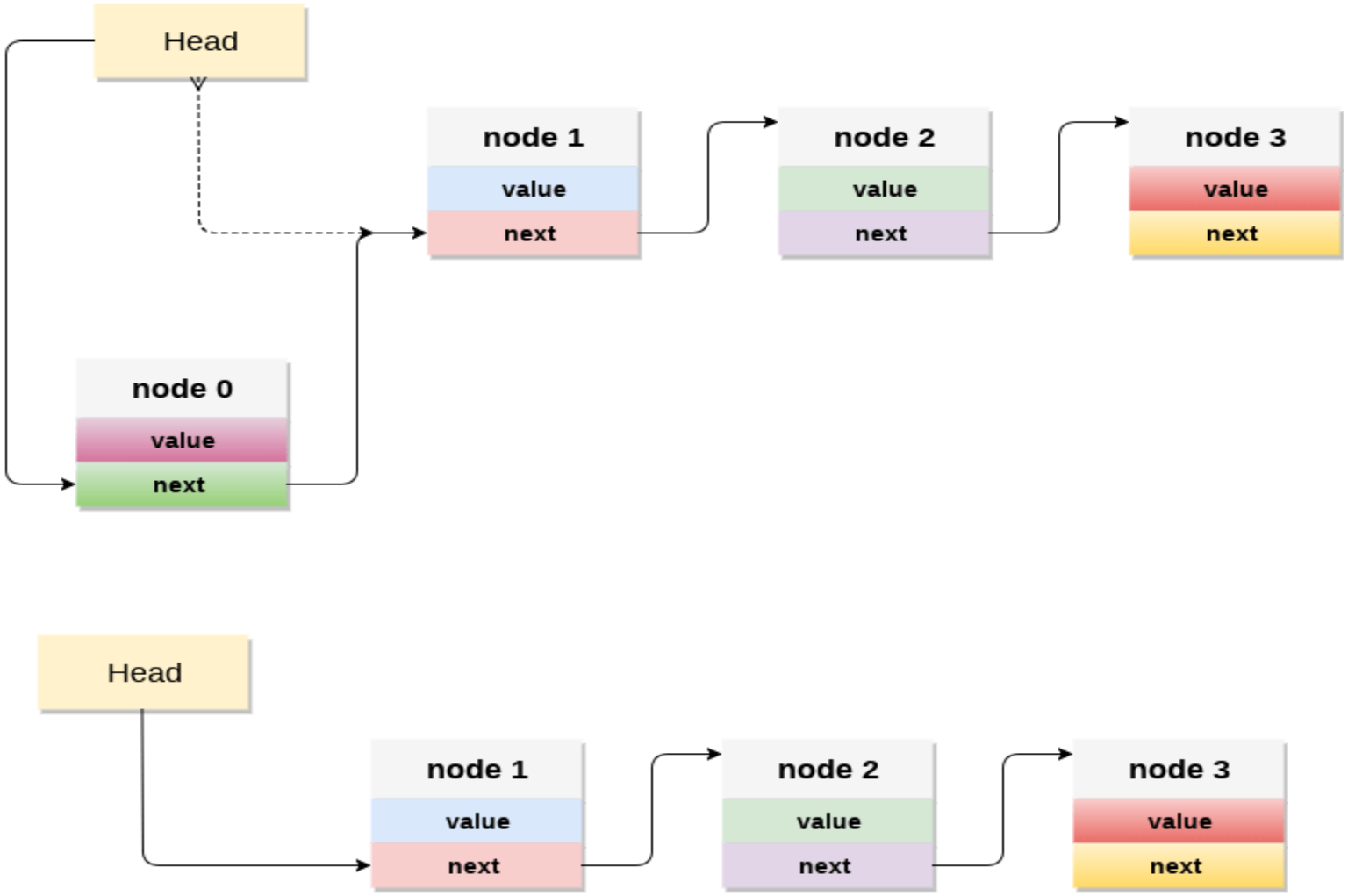
# PUSH Operation



# PUSH Operation

```
Algorithm push ()
{
   int val;
   Allocate memory to newnode;
   if(newnode == NULL)
   {
        Write not able to push the element;
   }
   else
   {
      read val;
      newnode ->data = val;
      newnode ->next = head;
       Top= newnode ;
     write Item pushed;
   }
}
```

# POP Operation



# POP Operation

```
Algorithm  POP()
{
     int item;
     struct node *temp;
     if (Top == NULL)
     {
          write Underflow;
      }
      else
       {
            item = Top->val;
            temp = Top;
            Top = Top->next;
              free(temp);
              write Item popped;

        }
}
```

# Traversing or Display

```
Algorithm display()
{
        struct node *ptr;
        temp=Top;
        if(top == NULL)
        {
             write Stack is empty;
        }
        else
        {
             write Printing Stack elements;
             while(temp!=NULL)
              {
                   write temp->val;
                    temp = temp->next;
               }
         }
}
```

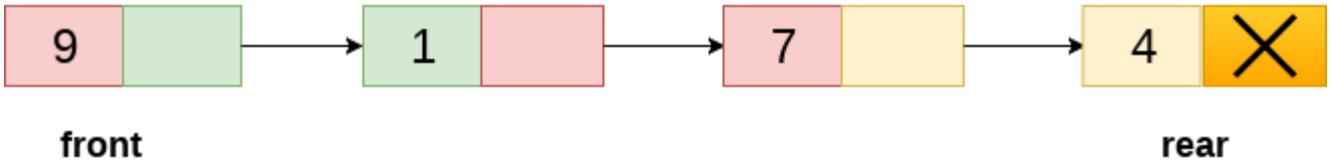
# Topics

- Stacks using linked List
- **Queues using Linked List**

# Linked List Implementation of Queues

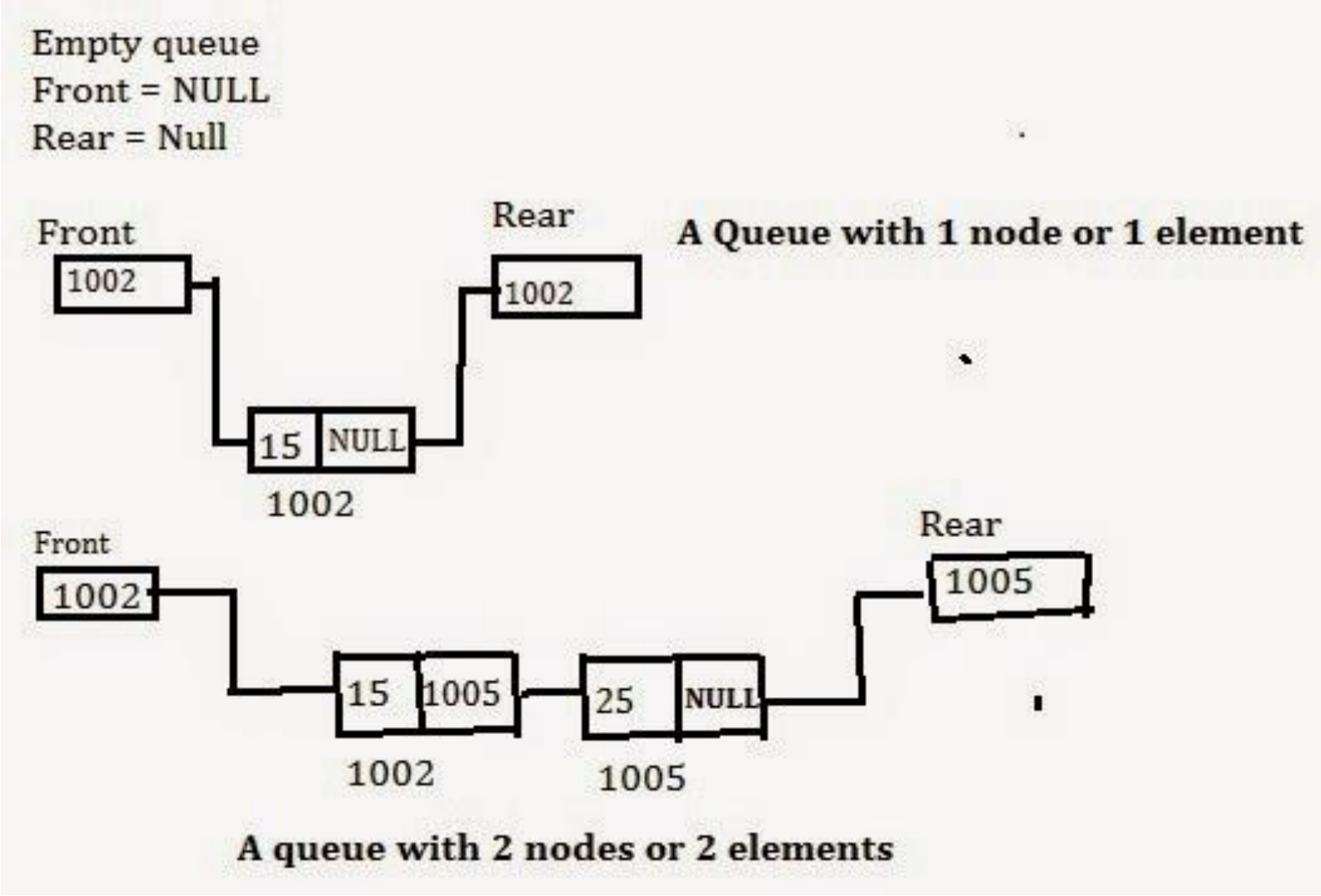
- In a linked queue, each node of the queue consists of two parts i.e. data part and the link part.
- Each element of the queue points to its immediate next element in the memory.
- In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer.
- The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.
- Insertion and deletions are performed at rear and front end respectively.
- If front and rear both are NULL, it indicates that the queue is empty.

# Linked List Implementation of Queues




Linked Queue

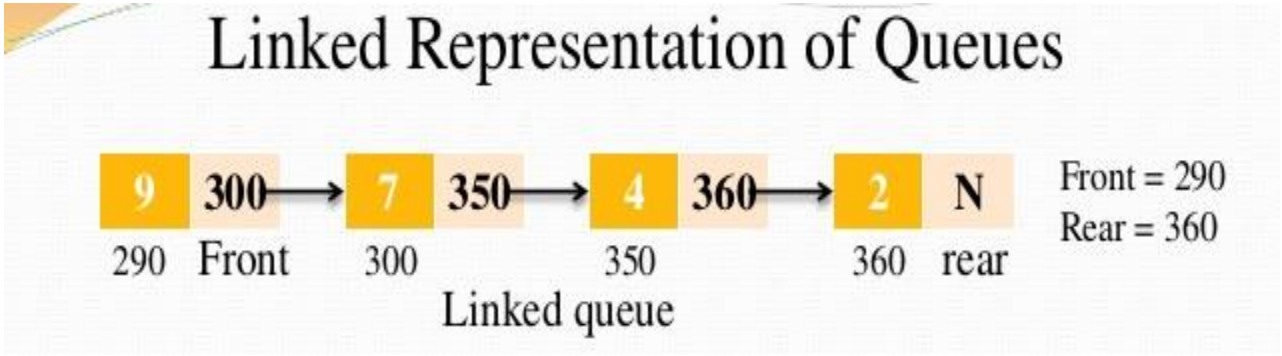
# Insert Operation

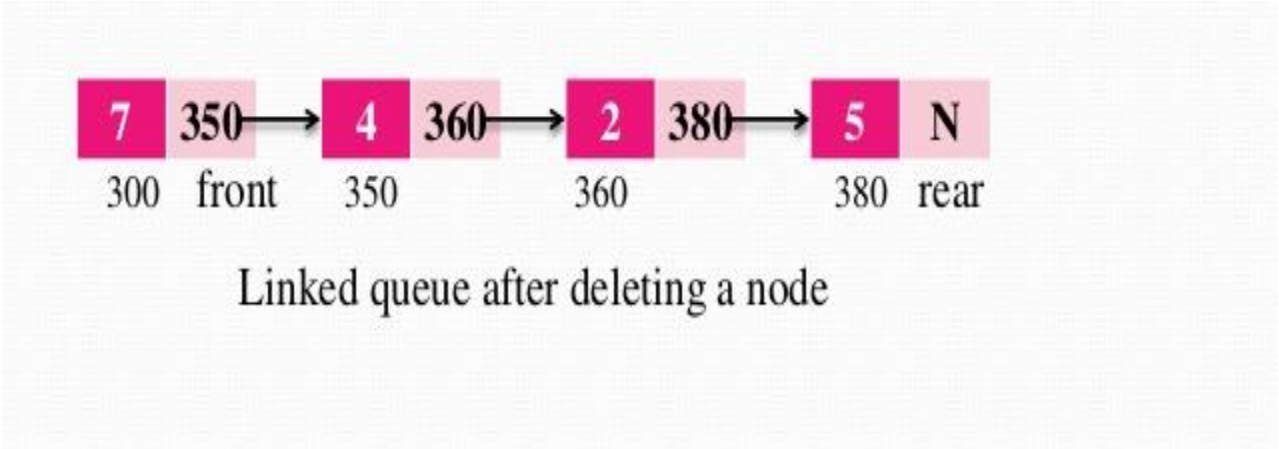


# Insert Operation

```
Algorithm Insert_Q()
{
  Struct node *newnode;
  Allocate memory to newnode;
   if(newnode == NULL)
   {
      write memory not allocated
      return;
   }
   else
   {
      newnode -> data = item;
      if(front == NULL)
      {
         front = newnode;
         rear = newnode;
         front -> next = NULL;
         rear -> next = NULL;
       }
      else
      {
         rear -> next =  newnode;
         rear =  newnode;
         rear->next = NULL;
      }
   }
}
```

# Delete OPeration



# Delete Operation

```
Algorithm delete ()
{
   struct node *ptr;
   if(front == NULL)
   {
      write UNDERFLOW;
      return;
   }
   else
   {
      ptr = front;
      front = front -> next;
      free(ptr);
   }
}
```

# Traversing or Display

```
Algorithm display()
{
    struct node *ptr;
    ptr = front;
    if(front == NULL)
    {
       Write Empty queue;
    }
    else
    {
        Write Queue Elements:;
        while(ptr != NULL)
        {
            Write ptr -> data;
            ptr = ptr -> next;
        }
    }
}
```