SRI MANAKULA VINAYAGAR ENGINEERING COLLEGE
(Approved by AICTE, New Delhi & Affiliated to Pondicherry University)
(Accredited by NBA-AICTE, New Delhi, ISO 9001:2000 Certified Institution & Accredited by NAAC with "A" Grade)
**(An Autonomous Institution)**
Madagadipet, Puducherry - 605 107

**DEPARTMENT OF CIVIL ENGINEERING**

Subject Name: **Data Structures**
Subject Code:

**Prepared by:**
Ms.V.SWATHILAKSHMI / AP /CSE
**Verified by:** **Approved by:**

## UNIT – III

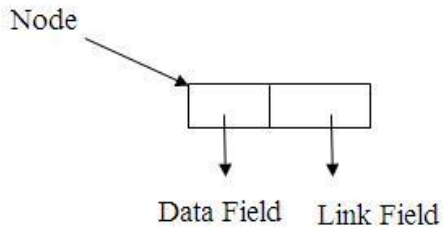
## LINKED LIST OPERATIONS

**Linked Lists**: Singly linked lists: Representation in memory, Algorithms of several operations: Traversing, Searching, Insertion, Deletion in linked list; Linked representation of Stack and Queue. Doubly linked list: Operations. Circular Linked Lists: operations.

### 2 MARKS

**1. What is Linked list? (Nov 11, Nov 13, Apr 15)**

- Linked list is a dynamic and linear data structure.
- Linked list is an ordered collection of elements in which each element is referred as a node.
- Each node has two fields namely
  i. Data field or Information field and
  ii. Address field or Link field.




**2. What are various fields in a Linked list?**

Each node has two fields namely
i. Data field or Information field and
ii. Address field or Link field.

**Data Field:** Data field contains the actual data.

**Address Field:** Address field contains the address of another node.

**3. Head Pointer?**

- Head pointer is the pointer which always points the first node in the list.
- Head pointer holds the address of the first node.
- Using Head pointer only we can move from first node to last node.

**4. Define External address, internal address and Null address?**

**External Address:** External address is an address stored in head pointer which holds the address of the firstnode.

**Internal Address:** Internal address is an address stored in link field of the each node except the last node. **Null Address:** Null address is an address stored in link field of the last node indicates the end of the list.

**5. What are the types of Linked List? (Apr 15)**

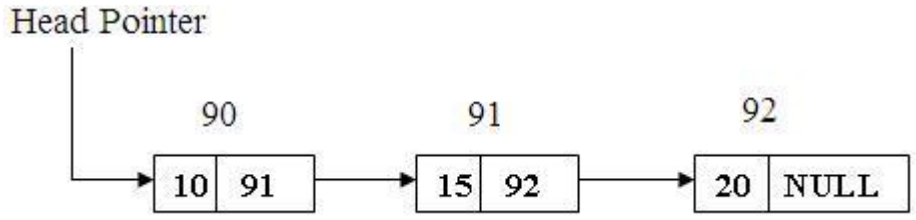
There are three types of Linked list. They are

i. Single Linked list,

ii. Double Linked list, and Circular Linked list.

**6. What is Single linked list?**

- In Single linked list, each node has one link to the next node.
- In Single linked list, we can move from only one direction from head pointer to Null pointer.
- Each node has two fields namely
  i. Data field or Information field and
  ii. Address field or Link field.
- It is otherwise called as Linear Linked list.
- Consider the following single linked list,




**7. What is Double Linked list? (Apr 12, Apr 13) (NOV 15)**

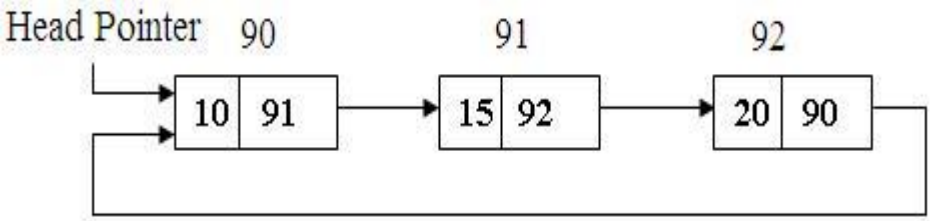
- Each node consist of three fields namely
  i. Previous address field or Backward link field,
  ii. Data field or Information field,
  iii. External address field or Forward link field.
- The previous address field holds the address of the previous node and the next address field holds the address of next node.
- In Double Linked list, we can move in both the direction from head pointer to Null address or vice versa.
- Consider the following linked list,

**8. What is Circular Linked list? Mention its types.**

- In Circular linked list, the last node is connected to the first node.
- In Circular Linked list, we can move from head pointer to Null address.
- There are two types of Circular linked list. They are
    i. Circular Single Linked list and
    ii. Circular Double Linked list.

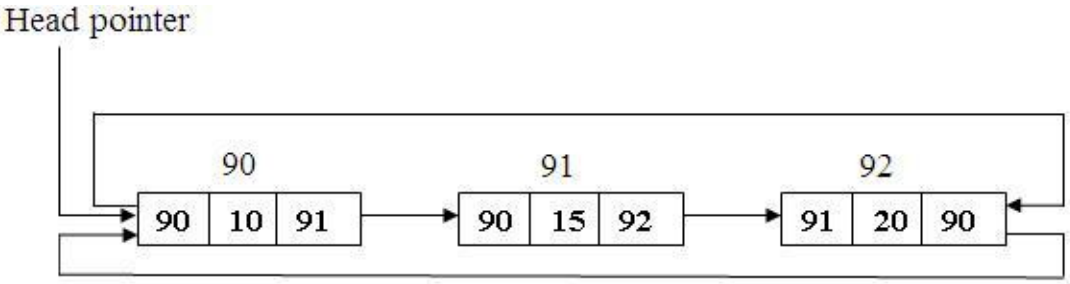
**9.What is Circular Single Linked list?**

- In Circular Single linked list, the last node is connected to the first node.
- In Circular Single linked list, we can move from only one direction from head pointer to Null pointer
- Consider the following circular single linked list,




**10. What is Circular Double Linked list?**

- In Circular Double linked list, the last node is connected to the first node.
- In Circular Double Linked list, we can move in both the direction from head pointer to Null address or vice versa. Consider the following circular single linked list,




**11.In how many ways we can insert a node into a single linked list?**

We can insert a new node into the list by following positions

i. Inserting a node as a first node,
ii. Inserting a node as a last node and
iii. Inserting a node as an intermediate node.

**12. How can we insert a new node as a first node?**

- If we want to insert a new node into a single linked list, first we need the empty node with empty data field and empty address field.
- Now store the new element into data field of the new node.
- To insert as a first node, the link field of the new node should replaced by the head pointer.
- Head pointer should replaced by the address of new node.

**13. How can we insert a new node as a last node?**

If we want to insert a new node into a single linked list, first we need the empty node with empty data field and empty address field.

- Now store the new element into data field of the new node.
- To insert as a last node, the link field of the last node should replaced by the address of new node.
- Link field of the new node should replaced by NULL.

**14. How can we insert a new node as an intermediate node?**

- If we want to insert a new node into a single linked list, first we need the empty node with empty data field and empty address field.
- Now store the new element into data field of the new node.
- To insert as an intermediate node, first we should know the address of the previous node. Then replace the link field of the previous node by the address of new node.
- Replace the link field of new node by the address of successor.

**15. How does a stack-linked list differ from a linked list?**

A stack linked list refers to a stack implemented using a linked list. That is to say, a linked list in which you can only add or remove elements to or from the top of the list. A stack-linked list accesses data last in, first out; a linked list accesses data first in, first out.

**16. How can you search for data in a linked list?**

The only way to search a linked list is with a linear search, because the only way a linked list's members can be accessed is sequentially. Sometimes it is quicker to take the data from a linked list and store it in a different data structure so that searches can be more efficient.

**17. List the basic operations carried out in a linked list.**

The basic operations carried out in a linked list include.

- Creation of list.
- Insertion of a node.
- Deletion of a node.
- Modification of a node.
- Traversal of the list.

**18. List the operations other than the basic operations that carried out in a linked list.**

The operations other than the basic operations that carried out in a linked list includes

- Searching an element in a list.
- Finding the predecessor element of a node.

- Finding the successor element of a node.
- Appending a linked list to another existing list.
- Splitting a linked list in to two lists.
- Arranging a linked list in ascending or descending order.

**19. List out the advantages in using a linked list. (Nov 14)**

The advantages in using a linked list are

- It is not necessary to specify the number if elements in a linked list during its declaration
- Linked list can grow and shrink in size del=pending upon the insertion and deletion that occurs in the list.
- Insertions and deletions at any place in a list can be handled easily and efficiently
- A linked list does not waste any memory space.

**20. List out the disadvantages in using a linked list.**

The advantages in using a linked list

- Searching a particular element in list is difficult and time consuming
- A linked list will use more storage space than an array to store the same number of elements.

**21. List out the application of a linked list**

Some of the important applications of linked lists are

- Manipulation of polynomials,
- Stacks and
- Queues.

**22. State the difference between arrays and linked list.**

| Arrays | Linked List |
|---|---|
| Size of any arrays is fixed. | Size of a list is variable. |
| It is necessary to specify the number of elements during the declaration. | It is not necessary to specify the number of elements during the declaration. |
| It occupies less memory space than linked list for the same memory number of elements. | It occupies more memory space. |

**23. How will you search an element in a linked list by iterative approach?**

Searching an element in a given linked list will happen sequentially starting from the head node of the list until the element has been found.

1) Initialize a node pointer, temp = head

2) Do following while temp is not NULL

- temp->data is equal to the data being searched, return true.
- temp = temp->next

3) Return false

**24. How will you search an element in a linked list by iterative approach?**

Searching an element in a given linked list will happen sequentially starting from the head node of the list until the element has been found.

1) If head is NULL, return false.

2) If head's data is same as the data to be searched, return true;

3) Else return search(head->next, search_data)

**25. How will you insert an node in a linked list?**

struct Node

{

int data;

struct Node *next;

};

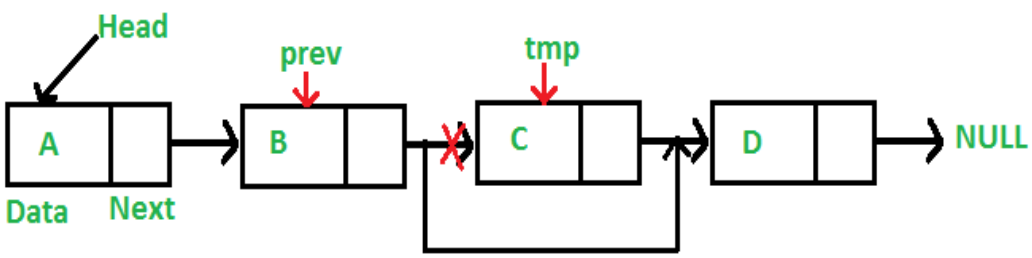
A node can be added in three ways

- At the front of the linked list
- After a given node.
- At the end of the linked list.

**26. How will you insert an node in a linked list?**

To delete a node from linked list, we need to do following steps.

1) Find previous node of the node to be deleted.
2) Change the next of previous node.
3) Free memory for the node to be deleted.



**27. Mention the advantages of representing stacks using linked list than arrays.**

The advantages of representing stacks using linked list than arrays are

- It is not necessary to specify the number of elements to be stored in a stack during its declaration.
- Insertions and deletions can be handled easily and efficiently.
- Linked list representation of stacks can grow & shrink in size without wasting the memory space, depending upon the insertion and deletion that occurs in the list.
- Multiple stacks can be represented efficiently using a chain for each stack.

**28. Mention the representation of DLL.**

- Doubly Linked List contains a link element called first and last.
- Each link carries a data field(s) and two link fields called next and prev.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.

**29. What are the operations performed in DLL?**

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Insert Last** − Adds an element at the end of the list.
- **Delete Last** − Deletes an element from the end of the list.
- **Insert After** − Adds an element after an item of the list.
- **Delete** − Deletes an element from the list using the key.
- **Display forward** − Displays the complete list in a forward manner.
- **Display backward** − Displays the complete list in a backward manner

**30. Advantages of circular linked list.**

- 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 queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of 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 to 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.

## 5 Marks

**1. What is Linked List? What are its types?**

Some demerits of array, leads us to use linked list to store the list of items. They are,

a. It is relatively expensive to insert and delete elements in an array.

*b.* Array usually occupies a block of memory space, one cannot simply double or triple the size of an array when additional space is required. *(For this reason, arrays are called **"dense lists"** andare said to be **"static"** data structures.)*

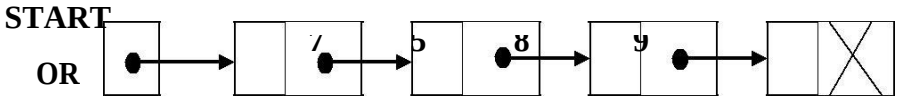
A ***linked list***, or ***one-way list***, is a linear collection of data elements, called ***nodes***, where the linear order is given by means of ***pointers***. That is, each node is divided into two parts:

- The first part contains the information of the element i.e. INFO or DATA.
- The second part contains the ***link field***, which contains the address of the next node in the list.
- The linked list consists of series of nodes, which are not necessarily adjacent in memory.
- A list is a ***dynamic data structure*** i.e. the number of nodes on a list may vary dramatically as elements are inserted and removed.

The pointer of the last node contains a special value, called the ***null*** pointer, which is any invalid address. This ***null pointer*** signals the end of list.

The list with no nodes on it is called the ***empty list*** or ***null list***.

***Example****: The linked list with 4 nodes.*




**Types of Linked Lists:**

a) Singly Linked List

b) Circular Linked List

c) Two-way or doubly linked lists

d) Circular doubly linked lists

**2. Write an advantage of linked list over Array?**

- An array is the data structure that contains a collection of similar type data elements whereas the Linked list is considered as non-primitive data structure contains a collection of unordered linked elements known as nodes.
- In the array the elements belong to indexes, i.e., if you want to get into the fourth element you have to write the variable name with its index or location within the square bracket.
- In a linked list though, you have to start from the head and work your way through until you get to the fourth element.
- Accessing an element in an array is fast, while Linked list takes linear time, so it is quite a bit slower.
- The requirement of memory is less due to actual data being stored within the index in the array. As against, there is a need for more memory in Linked Lists due to storage of additional next and previous referencing elements.

- Arrays are of fixed size. In contrast, Linked lists are dynamic and flexible and can expand and contract its size.

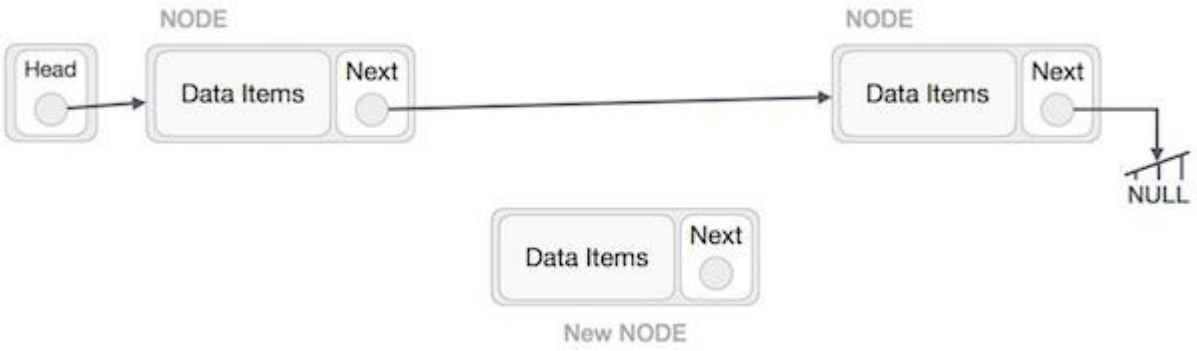
**3. Explained any two operations of Linked List?**

Following are the basic operations supported by a list.

- **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.

**INSERTION OPERATION:**

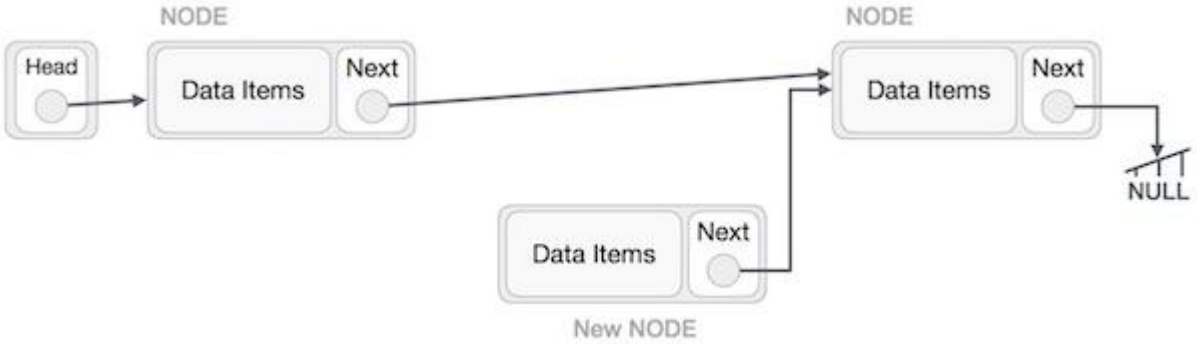
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.




Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C −

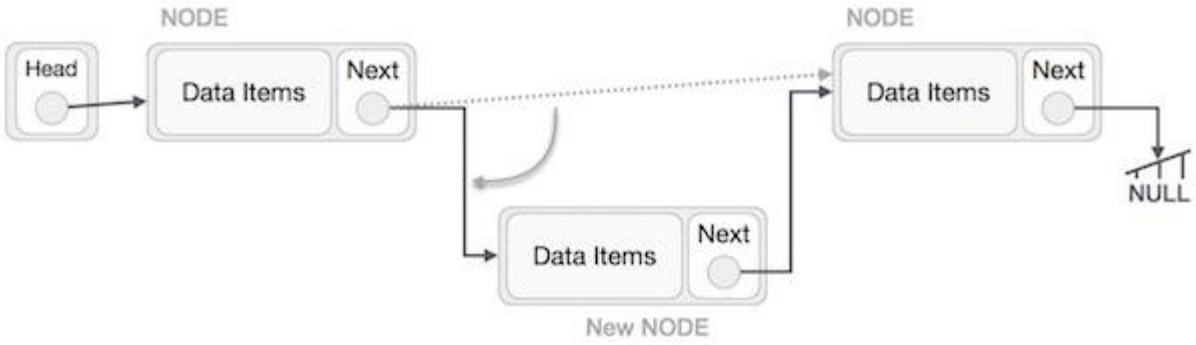
NewNode.next −>RightNode;

It should look like this −




Now, the next node at the left should point to the new node.

LeftNode.next −>NewNode;




This will put the new node in the middle of the two. The new list should look like this −

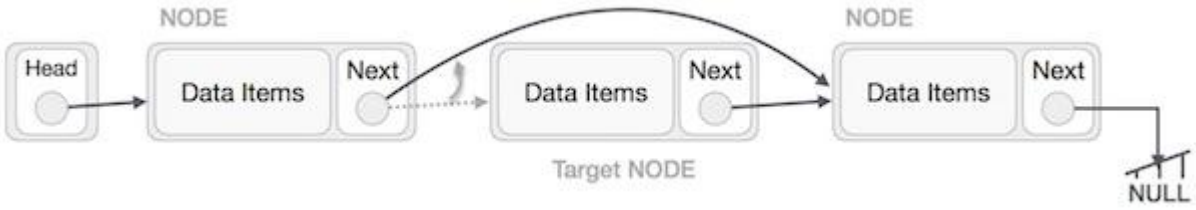

Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.'

**DELETE OPERATION:**

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.

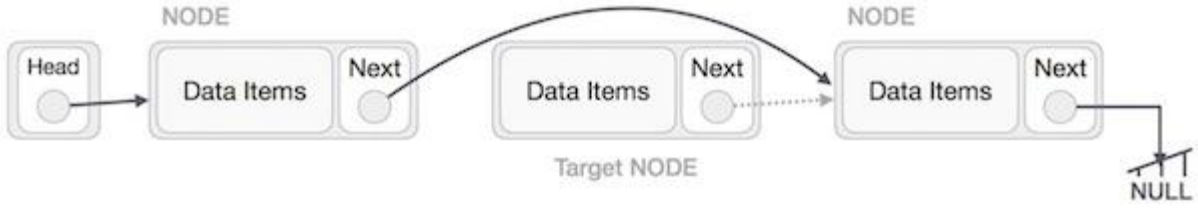

The left (previous) node of the target node now should point to the next node of the target node −

LeftNode.next −>TargetNode.next;

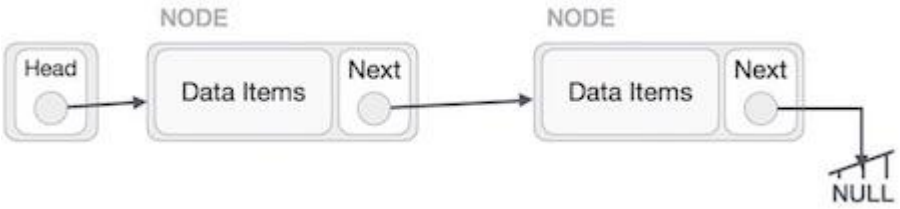



This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

TargetNode.next −> NULL;




We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

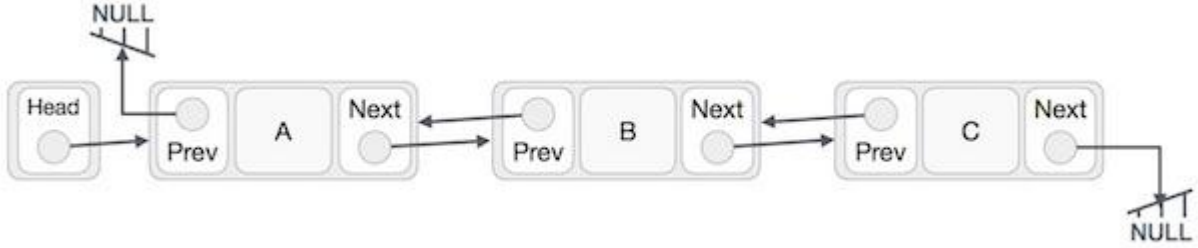



**4. Explained Doubly Linked List?**

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

- **Link** − Each link of a linked list can store a data called an element.
- **Next** − Each link of a linked list contains a link to the next link called Next.
- **Prev** − Each link of a linked list contains a link to the previous link called Prev.
- **LinkedList** − A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List Representation




**BASIC OPERATION:**

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Insert Last** − Adds an element at the end of the list.
- **Delete Last** − Deletes an element from the end of the list.
- **Insert After** − Adds an element after an item of the list.
- **Delete** − Deletes an element from the list using the key.
- **Display forward** − Displays the complete list in a forward manner.
- **Display backward** − Displays the complete list in a backward manner

5. **Explain insertion and deletion operations of doubly linked list?**

**INSERTION OPERATION:**

```
//insert link at the first location
void insertFirst(int key, int data) {

  //create a link
struct node *link = (struct node*) malloc(sizeof(struct node));
  link->key = key;
  link->data = data;

  if(isEmpty()) {
    //make it the last link
    last = link;
  } else {
    //update first prev link
    head->prev = link;
  }

  //point it to old first link
  link->next = head;

  //point first to new first link
  head = link;
}
```

**DELETION OPERATION:**

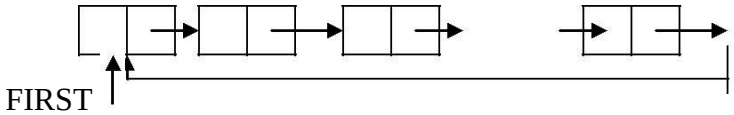
```
//delete first item
struct node* deleteFirst() {

  //save reference to first link
struct node *tempLink = head;

  //if only one link
  if(head->next == NULL) {
    last = NULL;
  } else {
    head->next->prev = NULL;
  }

  head = head->next;

  //return the deleted link
  return tempLink;
}
```

**6. Explain circular linked list?**

In the singly linked linear list the last node consist of the NULL pointer. Slightly improvement in this type of linked list is accomplished by replacing the null pointer in the last node of a list with the address of its first node.such a list is called circularly linked linear list or simply a circular list

7.

**STRUCTURE OF A CIRCULAR LIST:**




**ADVANTAGE OF CIRCULAR LIST OVER SINGLY LINKED LISTS*:***

1. Accessibility of a node from a given node, every node is easily accessible i.e., all node be reached by merely chaining through the list

2. Deletion operation: In addition to the address of x the node to be deleted from a singly list, it is also necessary to give address of the first node of the list in order to the predecessor of X.

Such a requirement does not exist for a circular list, since the search for the predecessor of node X can be initiated from X itself.

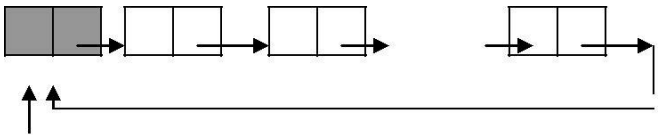
**DISADVANTAGE OF A CIRCULAR LIST:**

In processing a circular list, if we are not able to detect the end of the list, it is possible to get into an infinite loop.

This problem can be sowed i.e, the end of list can be detected by placing a special node which can be easily identified in the circular list .This special node is often called the list head of the circular list.

One more advantage of using this technique is that the list can never be empty which can be checked in the operation of singly linked list.

Representation of a circular list with a list head is shown below:




Here, variable HEAD denote the address of the list head .INFO field in the list head node is not used,which is shown by the shading the field.

An empty list is represented by having LINK(HEAD)=HEAD.

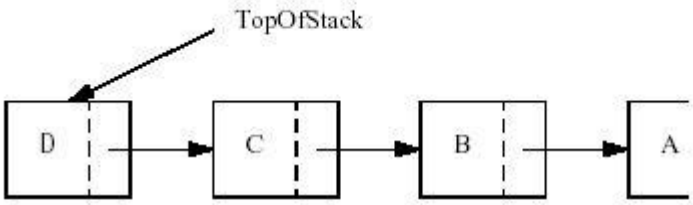
The algorithm for inserting a node at the head of a circular list with a list head consist of the following

steps:

NEW NODE
INFO(NEW) y
LINK(HEAD) LINK(H
EAD)
LINK(HEAD) NEW

**7. Describe about Linked Stack.**

The operation of adding an element to the front of a linked list is quite similar to that of pushing an element onto a stack. A stack can be accessed only through its top element, and a list can be accessed only from the pointer to its first element. Similar, the operation of removing the first element from a link list is analogous to popping a stack.




Linked list implementation of the stack

In both cases the only accessible item of collection is removed from that collection, and the next item becomes immediately accessible. The First node of the list is the top of the stack.

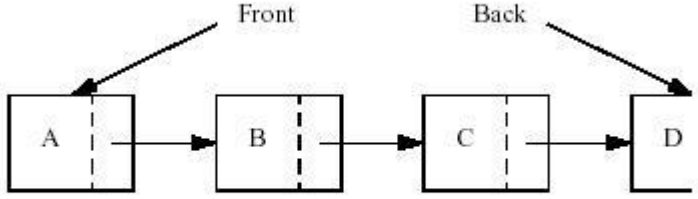
Available free memory spaces can be thought of as a finite pool of empty nodes existing initially. The most natural form from this pool to take that of a linked together by the next field in each node. This pool cannot be accessed by the programmer except through the memory allocation functions and free function. For eg: malloc function remove the first node from the list where as free return a mode to the front of the list. The list of available node is called the available list.

The advantage of the list implementation of the stack is that of all stacks being used by a program can share the same available list.

When any stack needs a node ,it can obtain it from the single variable list. When any stack no longer needs no node ,it returns the node to that same available list. As long as the total amount of spaces needed by all stack at any onetime is less than the amount of space initially available to them all ,each stack is available to grow and shrink to any size . No space has been preallocated to any single stack and stack is using the space that is does not need.

**8. Describe about Linked Queue?**

In queue, items are deleted from the front of a queue and inserted at the rear. Let a pointer to the first element of the list represent the front of the queue .Another pointer to the last element of the list represents the rear of the queue as shown in the following figure:




Linked list implementation of the queue

For this NodeQueue class, begin by declaring two instance variables itsFront and itsRear, each referring to a Node. In general, the queue's itsFront will always refer to the Node containing the first data value (if any) and the queue's itsRear will always refer to the Node containing the last data value (if any). An empty queue has null for itsFront, since there are no data values and so no Nodes at all.

**9. Write an advantages, disadvantages and applications of Linked List?**

**Advantages:**

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces the access time.

**Disadvantages:**

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

**Applications:**

- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don't need to know the size in advance.

## 10 Marks

**1. Describe Doubly Linked List in detail?**

A linked list is a sequence of data structures, which are connected together via links.

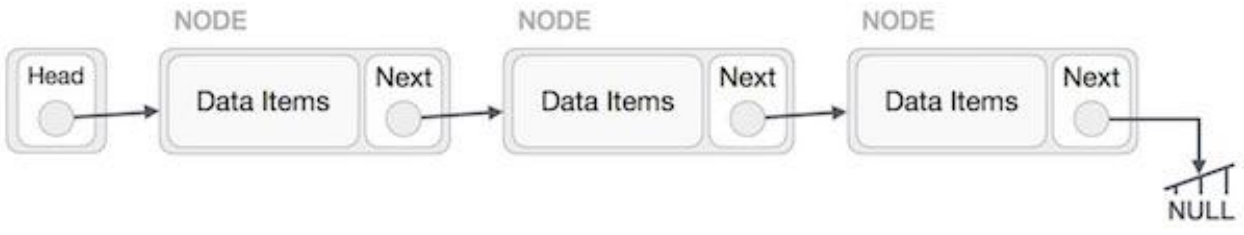
Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

**Link** − Each link of a linked list can store a data called an element.

**Next** − Each link of a linked list contains a link to the next link called Next.

**Linked List** − A Linked List contains the connection link to the first link called First

**Linked List Representation**



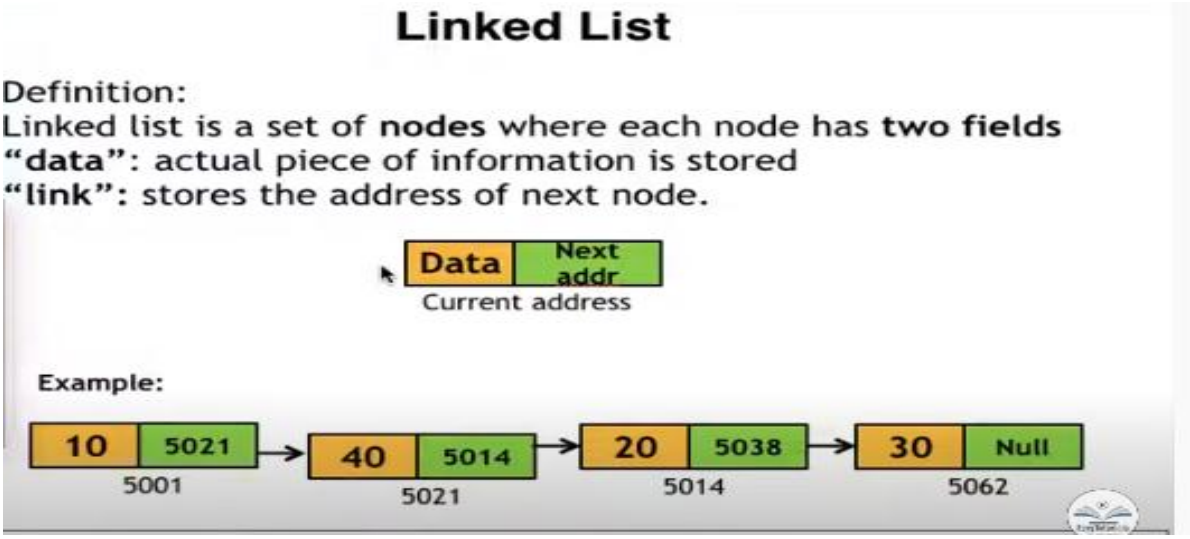
As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

## 2. What is linked list?

- A **singly linked list** is a type of linked list that is *unidirectional*, that is, it can be traversed in only one direction from head to the last node (tail).
- Each element in a linked list is called a **node**. A single node contains *data* and a pointer to the *next* node which helps in maintaining the structure of the list

The first node is called the **head**; it points to the first node of the list and helps us access every other element in the list. The last node, also sometimes called the **tail**, points to *NULL* which helps us in determining when the list ends.




## When to use Linked List?

- The number of nodes in a list is not fixed and can grow and shrink on demand.
- Any application which has to deal with an unknown number of objects will to need to use a linked list.

Types of linked list

1. Singly Linear Linked list
2. Singly Circular linked list
3. Doubly Linear Linked list
4. Doubly circular linked list

## Basic Operations

Following are the basic operations supported by a list.
**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.


## Singly Linked List

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
struct node
{
int data;
struct node *next;
}*head=NULL,*p,*t;
int main()
{
intch;
   void insert_beg();
   void insert_end();
intinsert_pos();
   void display();
   void delete_beg();
   void delete_end();
```

```
intdelete_pos();
   while(1)
   {
printf("\n\n---- Singly Linked List(SLL) Menu ----");
printf("\n1.Insert\n2.Display\n3.Delete\n4.Exit\n\n");
printf("Enter your choice(1-4):");
scanf("%d",&ch);


      switch(ch)
      {
         case 1:
printf("\n---- Insert Menu ----");
printf("\n1.Insert at beginning\n2.Insert at end\n3.Insert at specified position\n4.Exit");
printf("\n\nEnter your choice(1-4):");
scanf("%d",&ch);


                switch(ch)
                {
                   case 1: insert_beg();
                           break;
                   case 2: insert_end();
                           break;
                   case 3: insert_pos();
                           break;
                   case 4: exit(0);
                   default: printf("Wrong Choice!!");
                }
                break;
```

```
        case 2: display();
                break;
        case 3: printf("\n---- Delete Menu ----");
        printf("\n1.Delete from beginning\n2.Delete from end\n3.Delete from specified position\n4.Exit");
printf("\n\nEnter your choice(1-4):");
scanf("%d",&ch);
                switch(ch)
                {
                   case 1: delete_beg();
                           break;
                   case 2: delete_end();
                           break;
                   case 3: delete_pos();
                           break;
                   case 4: exit(0);
                   default: printf("Wrong Choice!!");
                }
                break;
        case 4: exit(0);
                default: printf("Wrong Choice!!");
      }
   }
   return 0;
}

void insert_beg()
{
intnum;
```

```
    t=(struct node*)malloc(sizeof(struct node));
printf("Enter data:");
scanf("%d",&num);
    t->data=num;
    if(head==NULL)        //If list is empty
    {
        t->next=NULL;
        head=t;
    }
    else
    {
        t->next=head;
        head=t;
    }
}

void insert_end()
{
intnum;
    t=(struct node*)malloc(sizeof(struct node));
printf("Enter data:");
scanf("%d",&num);
    t->data=num;
    t->next=NULL;

    if(head==NULL)        //If list is empty
    {
        head=t;
```

```
    }
    else
    {
        p=head;
        while(p->next!=NULL)
        p=p->next;
        p->next=t;
    }
}

intinsert_pos()
{
intpos,i,num;
    if(head==NULL)
    {
printf("List is empty!!");
        return 0;
    }

    t=(struct node*)malloc(sizeof(struct node));
printf("Enter data:");
scanf("%d",&num);
printf("Enter position to insert:");
scanf("%d",&pos);
    t->data=num;

    p=head;
    for(i=1;i<pos-1;i++)
```

```
    {
        if(p->next==NULL)
        {
printf("There are less elements!!");
            return 0;
        }

        p=p->next;
    }

    t->next=p->next;
    p->next=t;
    return 0;
}

void display()
{
    if(head==NULL)
    {
printf("List is empty!!");
    }
    else
    {
        p=head;
printf("The linked list is:\n");
        while(p!=NULL)
        {
printf("%d->",p->data);
```

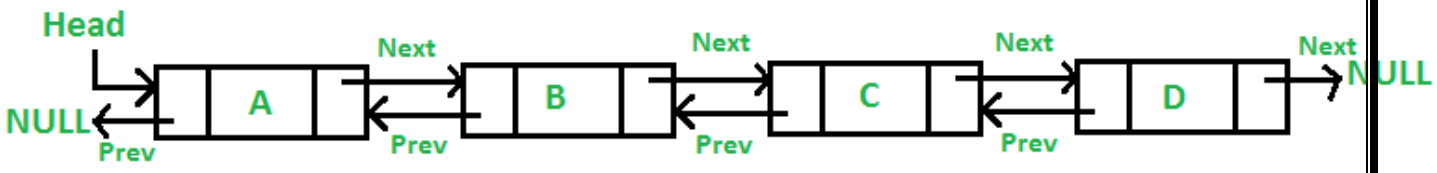
```
            p=p->next;
        }
    }
}

void delete_beg()
{
    if(head==NULL)
    {
printf("The list is empty!!");
    }
    else
    {
        p=head;
        head=head->next;
printf("Deleted element is %d",p->data);
        free(p);
    }
}

void delete_end()
{
    if(head==NULL)
    {
printf("The list is empty!!");
    }
    else
    {
```

```
        p=head;
        while(p->next->next!=NULL)
        p=p->next;

        t=p->next;
        p->next=NULL;
printf("Deleted element is %d",t->data);
        free(t);
    }
}

intdelete_pos()
{
intpos,i;

    if(head==NULL)
    {
printf("List is empty!!");
        return 0;
    }

printf("Enter position to delete:");
scanf("%d",&pos);

    p=head;
    for(i=1;i<pos-1;i++)
    {
        if(p->next==NULL)
```

```
        {
printf("There are less elements!!");
            return 0;
        }
        p=p->next;
    }

    t=p->next;
    p->next=t->next;
printf("Deleted element is %d",t->data);
    free(t);

    return 0;
}
```

**2. Describe Doubly Linked List in detail?**

A **D**oubly **L**inked **L**ist (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.




Following is representation of a DLL node in C language:

```
/* Node of a doubly linked list */
struct Node {
    int data;
    struct Node* next; // Pointer to next node in DLL
    struct Node* prev; // Pointer to previous node in DLL
```

};

**Advantages over singly linked list**

**1)** A DLL can be traversed in both forward and backward direction.
**2)** The delete operation in DLL is more efficient if pointer to the node to be deleted is given.
**3)** We can quickly insert a new node before a given node.
In singly linked list, to delete a node, 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 previous pointer.

**Disadvantages over singly linked list**

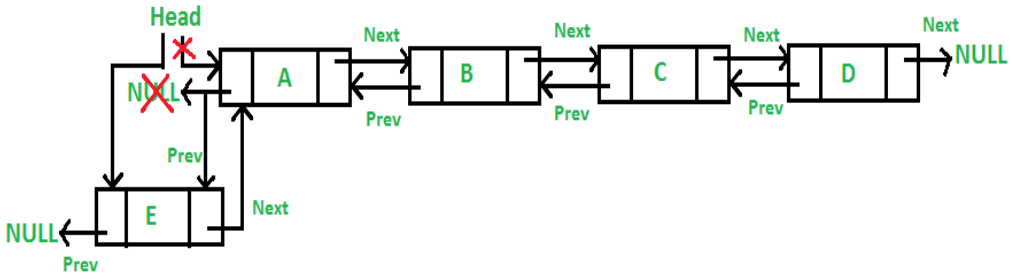
**1)** every node of DLL Require extra space for an previous pointer. It is possible to implement DLL with single pointer
**2)** all operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with next pointers. For example in following functions for insertions at different positions, we need 1 or 2 extra steps to set previous pointer.

**Insertion**

a node can be added in four ways
**1)** at the front of the DLL
**2)** after a given node.
**3)** At the end of the DLL
**4)** before a given node.

**Add a node at the front: (A 5 steps process)**

The new node is always added before the head of the given Linked List. And newly added node becomes the new head of DLL. For example if the given Linked List is 10152025 and we add an item 5 at the front, then the Linked List becomes 510152025. Let us call the function that adds at the front of the list is push (). The push () must receive a pointer to the head pointer, because push must change the head pointer to point to the new node




/* Given a reference (pointer to pointer) to the head of a list

and an int, inserts a new node on the front of the list. */

void push(struct Node** head_ref, intnew_data)

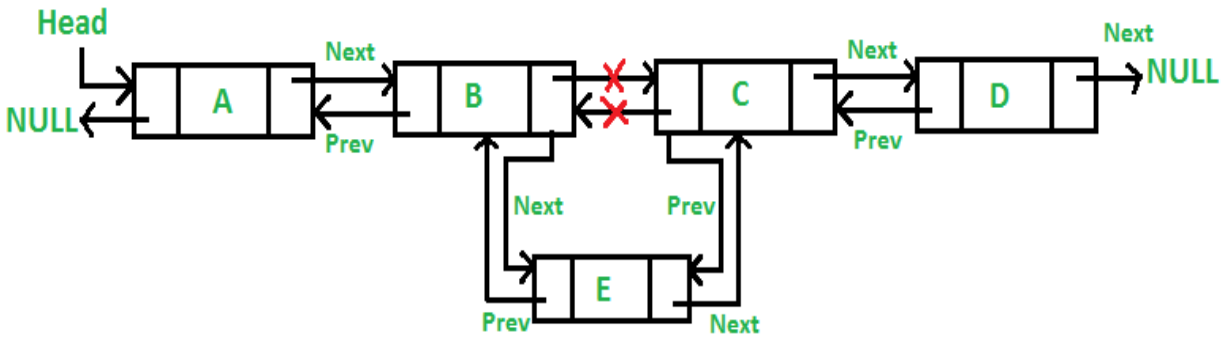
{

/* 1. allocate node */

struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

```
    /* 2. put in the data */
    new_node->data = new_data;
    /* 3. Make next of new node as head and previous as NULL */
    new_node->next = (*head_ref);
    new_node->prev = NULL;


    /* 4. change prev of head node to new node */
    if ((*head_ref) != NULL)
            (*head_ref)->prev = new_node;
    /* 5. move the head to point to the new node */
    (*head_ref) = new_node;
}
```

**Add a node after a given node.: (A 7 steps process)**
We are given pointer to a node as prev_node, and the new node is inserted after the given node.



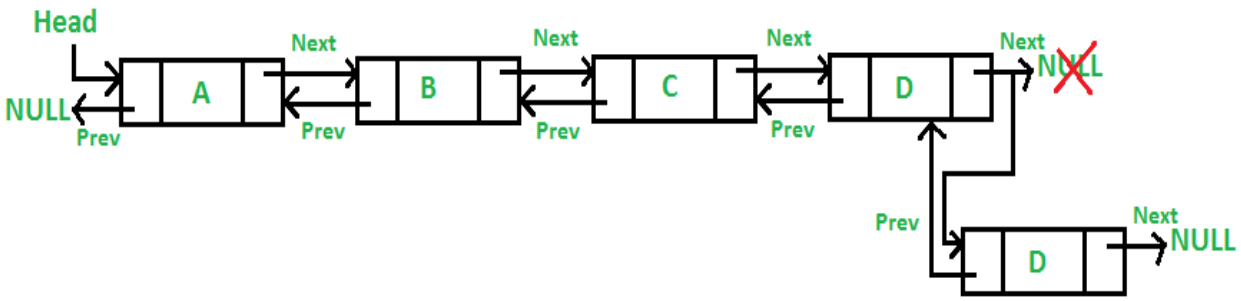

```
/* Given a node as prev_node, insert a new node after the given node */
void insertAfter(struct Node* prev_node, intnew_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL) {
            printf("the given previous node cannot be NULL");
            return;
    }
```

```
    /* 2. allocate new node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    /* 3. put in the data */
    new_node->data = new_data;

    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;

    /* 5. Make the next of prev_node as new_node */
    prev_node->next = new_node;

    /* 6. Make prev_node as previous of new_node */
    new_node->prev = prev_node;

    /* 7. Change previous of new_node's next node */
    if (new_node->next != NULL)
        new_node->next->prev = new_node;
}
```

**Add a node at the end: (7 steps process)**

The new node is always added after the last node of the given Linked List. For example if the given DLL is 510152025 and we add an item 30 at the end, then the DLL becomes 51015202530.

Since a Linked List is typically represented by the head of it, we have to traverse the list till end and then change the next of last node to new node



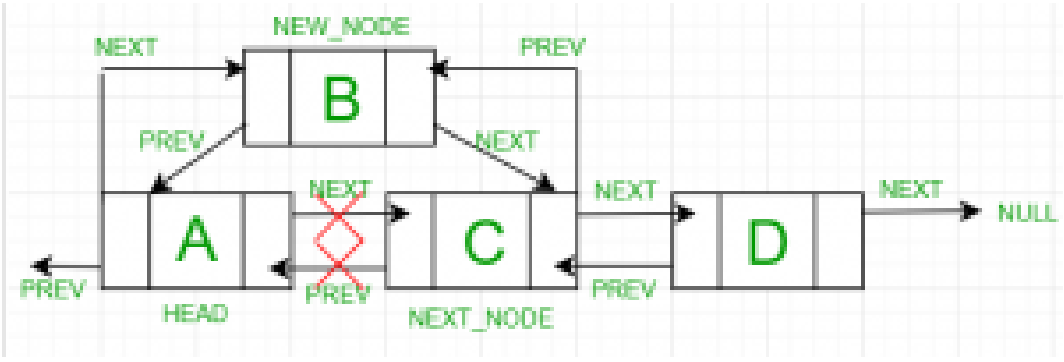

```
/* Given a reference (pointer to pointer) to the head
of a DLL and an int, appends a new node at the end */
void append(struct Node** head_ref, intnew_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    struct Node* last = *head_ref; /* used in step 5*/

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. This new node is going to be the last node, so
          make next of it as NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make the new
          node as head */
    if (*head_ref == NULL) {
          new_node->prev = NULL;
          *head_ref = new_node;
          return;
    }
```

```
    /* 5. Else traverse till the last node */
    while (last->next != NULL)
            last = last->next;
    /* 6. Change the next of last node */
    last->next = new_node;


    /* 7. Make last node as previous of new node */
    new_node->prev = last;
    return;
}
```

**Add a node before a given node:**

**Steps**
Let the pointer to this given node be next_node and the data of the new node to be added as new_data.

1. Check if the next_node is NULL or not. If it's NULL, return from the function because any new node can not be added before a NULL
2. Allocate memory for the new node, let it be called new_node
3. Set new_node->data = new_data
4. Set the previous pointer of this new_node as the previous node of the next_node, new_node->prev = next_node->prev
5. Set the previous pointer of the next_node as the new_node, next_node->prev = new_node
6. Set the next pointer of this new_node as the next_node, new_node->next = next_node;
7. If the previous node of the new_node is not NULL, then set the next pointer of this previous node as new_node, new_node->prev->next = new_node
8. Else, if the prev of new_node is NULL, it will be the new head node. So, make (*head_ref) = new_node



```
// A complete working C program to demonstrate all
// insertion before a given node
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};
/* Given a reference (pointer to pointer) to the head of a list
and an int, inserts a new node on the front of the list. */
void push(struct Node** head_ref, intnew_data)
{
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    new_node->data = new_data;

    new_node->next = (*head_ref);
    new_node->prev = NULL;

    if ((*head_ref) != NULL)
            (*head_ref)->prev = new_node;

    (*head_ref) = new_node;
}
```

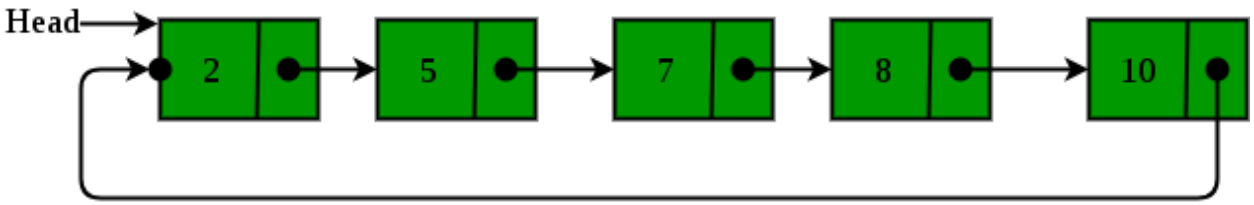
```
/* Given a node as next_node, insert a new node before the given node */
void insertBefore(struct Node** head_ref, struct Node* next_node, intnew_data)
{
    /*1. check if the given next_node is NULL */
    if (next_node == NULL) {
            printf("the given next node cannot be NULL");
            return;
    }

    /* 2. allocate new node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    /* 3. put in the data */
    new_node->data = new_data;

    /* 4. Make prev of new node as prev of next_node */
    new_node->prev = next_node->prev;

    /* 5. Make the prev of next_node as new_node */
    next_node->prev = new_node;

    /* 6. Make next_node as next of new_node */
    new_node->next = next_node;

    /* 7. Change next of new_node's previous node */
    if (new_node->prev != NULL)
            new_node->prev->next = new_node;
    /* 8. If the prev of new_node is NULL, it will be
```

```
    the new head node */
    else
        (*head_ref) = new_node;
}

// This function prints contents of linked list starting from the given node
void printList(struct Node* node)
{
    struct Node* last;
    printf("\nTraversal in forward direction \n");
    while (node != NULL) {
        printf(" %d ", node->data);
        last = node;
        node = node->next;
    }

    printf("\nTraversal in reverse direction \n");
    while (last != NULL) {
        printf(" %d ", last->data);
        last = last->prev;
    }
}

/* Driver program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;
```

```
    push(&head, 7);

    push(&head, 1);

    push(&head, 4);

    // Insert 8, before 1. So linked list becomes 4->8->1->7->NULL

    insertBefore(&head, head->next, 8);

    printf("Created DLL is: ");

    printList(head);

    getchar();

    return 0;

}
```

**3. Describe about Circular Linked List in detail.**

**Circular linked list** is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.




**Advantages of Circular Linked Lists:**

**1)** 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.

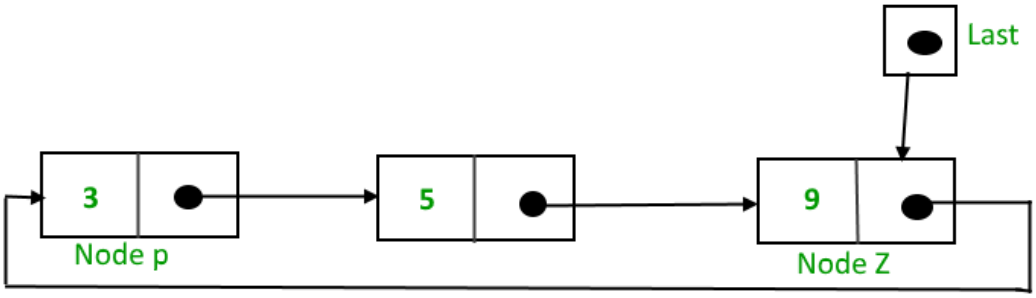
**2)** Useful for implementation of queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.

**3)** 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 to 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.

**4)** Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap

**Implementation**

To implement a circular singly linked list, we take an external pointer that points to the last node of the list. If we have a pointer last pointing to the last node, then last -> next will point to the first node.



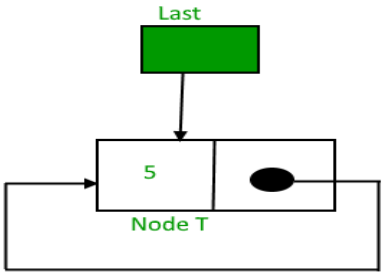

**Insertion**

A node can be added in three ways:

- Insertion in an empty list
- Insertion at the beginning of the list
- Insertion at the end of the list
- Insertion in between the nodes

Insertion in an empty List:

Initially when the list is empty, last pointer will be NULL.




After insertion, T is the last node so pointer *last* points to node T. And Node T is first and last node, so T is pointing to itself.

Function to insert node in an empty List,

```
struct Node *addToEmpty(struct Node *last, int data)

{

    // This function is only for empty list

    if (last != NULL)

    return last;
```

```
    // Creating a node dynamically.
    struct Node *last =
            (struct Node*)malloc(sizeof(struct Node));

    // Assigning the data.
    last -> data = data;

    // Note : list was empty. We link single node
    // to itself.
    last -> next = last;
    return last;
}
```

**Insertion at the beginning of the list**

To Insert a node at the beginning of the list, follow these step:

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

Function to insert node in the beginning of the List,

```
struct Node *addBegin(struct Node *last, int data)
{
if (last == NULL)
    return addToEmpty(last, data);

// Creating a node dynamically.
struct Node *temp
        = (struct Node *)malloc(sizeof(struct Node));

// Assigning the data.
temp -> data = data;
```

```
// Adjusting the links.

temp -> next = last -> next;

last -> next = temp;


return last;

}
```

**Insertion at the end of the list**

To Insert a node at the end of the list, follow these step:

1. Create a node, say T.
2. Make T -> next = last -> next;
3. last -> next = T.
4. last = T.

Function to insert node in the end of the List,

```
struct Node *addEnd(struct Node *last, int data)

{

if (last == NULL)

    return addToEmpty(last, data);


// Creating a node dynamically.

struct Node *temp =

            (struct Node *)malloc(sizeof(struct Node));


// Assigning the data.

temp -> data = data;


// Adjusting the links.

temp -> next = last -> next;

last -> next = temp;

last = temp;
```

```
return last;

}
```

**Insertion in between the nodes**

To Insert a node at the end of the list, follow these step:

1. Create a node, say T.
2. Search the node after which T need to be insert, say that node be P.
3. Make T -> next = P -> next;
4. P -> next = T.

Function to insert node in the end of the List,

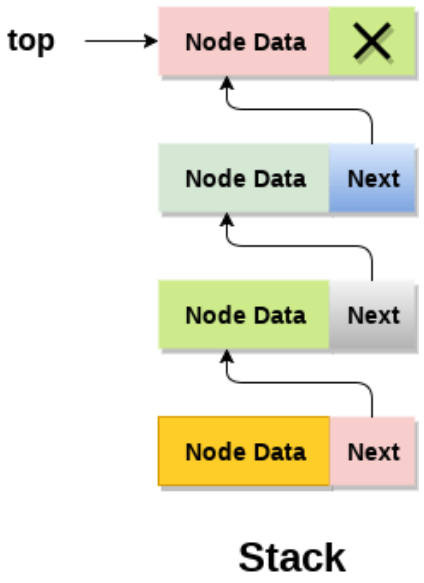
```
struct Node *addAfter(struct Node *last, int data, int item)

{

    if (last == NULL)

    return NULL;


    struct Node *temp, *p;

    p = last -> next;


    // Searching the item.

    do

    {

            if (p ->data == item)

            {

                    // Creating a node dynamically.

                    temp = (struct Node *)malloc(sizeof(struct Node));


                    // Assigning the data.

                    temp -> data = data;


                    // Adjusting the links.

                    temp -> next = p -> next;
```

```
                    // Adding newly allocated node after p.
                    p -> next = temp;
                    // Checking for the last node.
                    if (p == last)
                            last = temp;
                    return last;
                }
                p = p -> next;
        } while (p != last -> next);


        cout<< item << " not present in the list." <<endl;
        return last;
    }
```

**4. Describe about Linked Stack in detail.**

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

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.



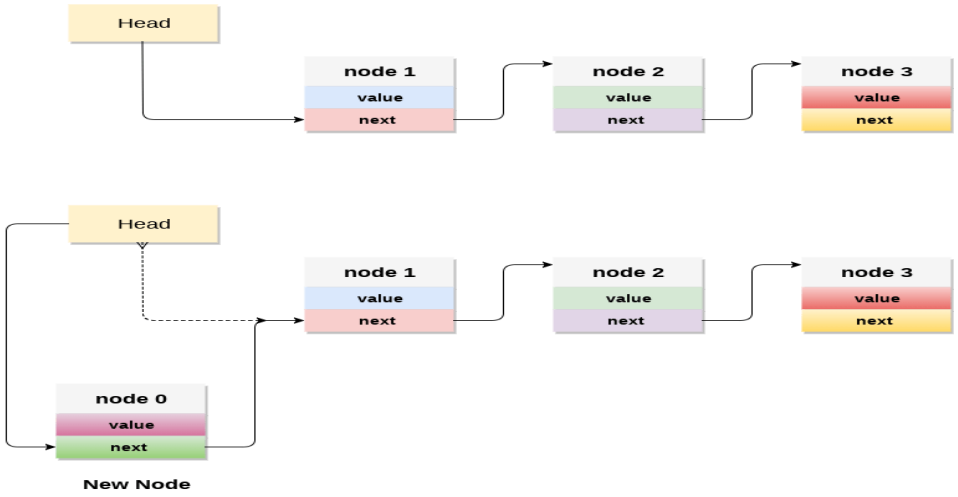
The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

Adding a node to the stack (Push operation)

Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

**Time Complexity : o(1)**




```
void push ()

{

intval;

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");
    }
}
```

Deleting a node from the stack (POP operation)

Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :

- **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
- **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

**Time Complexity : o(n)**

```
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");
    }
}
```

Display the nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

- Copy the head pointer into a temporary pointer.
- Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

**Time Complexity : o(n)**

```
void display()
{
inti;
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;
        }
    }
}
```

**4. Describe about Linked Queue in details.**

Due to the drawbacks of the array implementation of queue, the array implementation can not be used for the large scale applications where the queues are implemented. One of the alternative of array implementation is linked list implementation of queue.

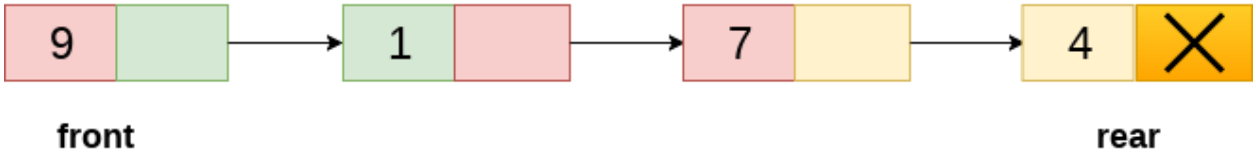
The storage requirement of linked representation of a queue with n elements is o(n) while the time requirement for operations is o(1).

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.

The linked representation of queue is shown in the following figure.




Linked Queue

**Operation on Linked Queue**

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

**Insert operation**

The insert operation append the queue by adding an element to the end of the queue. The new element will be the last element of the queue.

Firstly, allocate the memory for the new node ptr by using the following statement.

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

There can be the two scenario of inserting this new node ptr into the linked queue.

In the first scenario, we insert element into an empty queue. In this case, the condition **front = NULL** becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.

```
ptr -> data = item;
    if(front == NULL)
    {
       front = ptr;
       rear = ptr;
       front -> next = NULL;
       rear -> next = NULL;
    }
```

In the second case, the queue contains more than one element. The condition front = NULL becomes false. In this scenario, we need to update the end pointer rear so that the next pointer of rear will point to the new node ptr. Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node **ptr**. We also need to make the next pointer of rear point to NULL.

```
rear -> next = ptr;
       rear = ptr;
       rear->next = NULL;
```

**Algorithm**

- **Step 1:** Allocate the space for the new node PTR
- **Step 2:** SET PTR -> DATA = VAL

- **Step 3:** IF FRONT = NULL
  SET FRONT = REAR = PTR
  SET FRONT -> NEXT = REAR -> NEXT = NULL
  ELSE
  SET REAR -> NEXT = PTR
  SET REAR = PTR
  SET REAR -> NEXT = NULL
  [END OF IF]
- **Step 4:** END

```
void insert(struct node *ptr, int item; )

{

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

   if(ptr == NULL)

   {

printf("\nOVERFLOW\n");

      return;

   }

   else

   {

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;
        }
    }
}
```

**Deletion**

**Deletion operation** removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make exit.

Otherwise, we will delete the element that is pointed by the pointer front. For this purpose, copy the node pointed by the front pointer into the pointer ptr. Now, shift the front pointer, point to its next node and free the node pointed by the node ptr. This is done by using the following statements.

```
ptr = front;
    front = front -> next;
    free(ptr);
```

Algorithm

- **Step 1:** IF FRONT = NULL
  Write " Underflow "
  Go to Step 5
  [END OF IF]
- **Step 2:** SET PTR = FRONT
- **Step 3:** SET FRONT = FRONT -> NEXT
- **Step 4:** FREE PTR
- **Step 5:** END

```
void delete (struct node *ptr)
{
   if(front == NULL)
   {
printf("\nUNDERFLOW\n");
      return;
   }
   else
   {
ptr = front;
      front = front -> next;
      free(ptr);
   }
}
```

**5. Explain Single Linked List with its operations.**

**SINGLY / LINEAR LINKED LIST:**

In a sequential representation, suppose that the items were implicitly ordered, that is, each item contained within itself the address of the next item, such an implicit ordering gives rise to a data structure known as a Linear linked list or Singly linked list which is shown in figure:

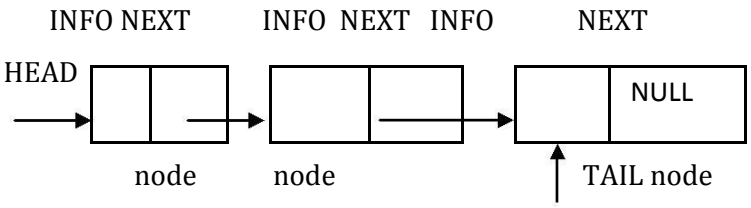
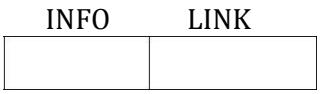



FIG:linear linked list.

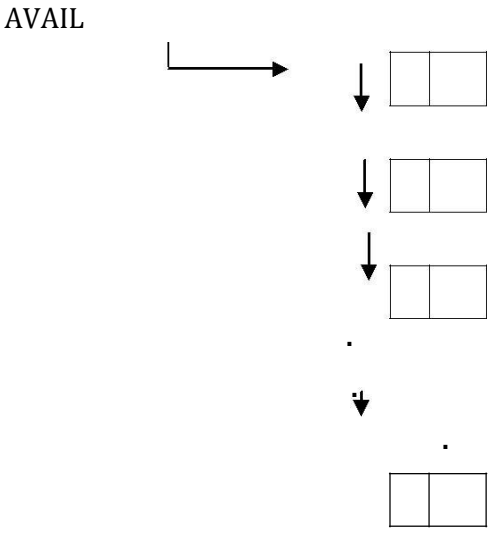
Each node consists of two fieds,

- information field called INFO that holds the actual element on the list and
- a pointer pointing to next element of the list called LINK ,ie. It holds the address of the next element.

The name of a typical element is denoted by NODE. Pictorially, the node NODE




Structure is given as follows It is assumed that an available area of storage for this node structure consists of a stack of available nodes,as shown below:




Here, the pointer variable AVAIL contains the address of the top node in the stack. The head and tail are pointers pointing to first and last element of the list repectively. For an empty list the head and tail have the value NIL. When the list has one element, the head and tail point to the same.

**OPERATIONS**:INSER

TION IN A LIST:

Inserting a new item,say 'x' ,into the list has three situations:

1. Insertion at the front of the list
2. Insertion in the middle of the list or in the order
3. Insertion at the end of the list

INSERTION AT FRONT:

Algorithms for placing the new item at the beginning of a linked list:

1. Obtain space for new node
2. Assign data to the item field of new node
3. Set the next field of the new node to point to the start of the list
4. Change the head pointer to point to the new node.

**Function** INSERT(X, FIRST)

**Variables used:**

X ← New element to be inserted

FIRST ← Pointer to the first element whose node contains INFO and LINK fields.

AVAIL ← Pointer to the top element of the availability stack

NEW ← Temporary pointer variable

1. [Underflow?] If

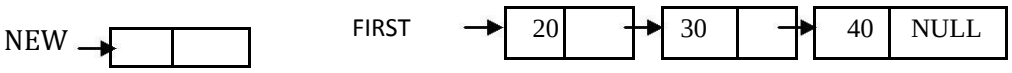
AVAIL = NULL

Then Write('AVAILABILITY STACK UNDERFLOW')

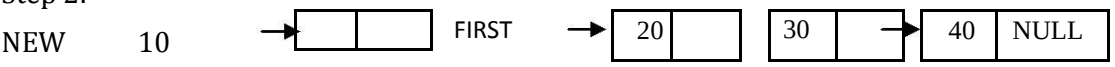
Return(FIRST)

2. [Obtain address of next free node]
   NEW ← AVAIL
3. [Remove free node from availability stack]
   AVAIL ← LINK(AVAIL)
4. [Initialize fields of new node and its link to the list]
   INFO(NEW) ← X
   LINK(NEW) ← FIRST
5. [Return address of new node]
   Return(NEW)

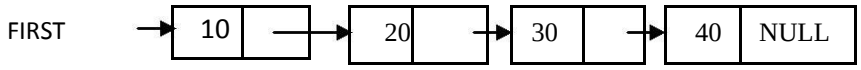
Step 1:




Step 2:




Step 3:




**Algorithm for inserting the new node x between the two existing nodes, say N1 and N2( or in order):**

1. Set space for new node x
2. Assign value to the item field of x
3. Search for predecessor node n1 of x
4. Set the next field of x to point to link of node n1(node N2)
5. Set the next field of N1 to point to x.

**Function** INSORD(X, FIRST)

**Variables used:**

X ← new element

FIRST ← Pointer to the first element whose node contains INFO and LINK fields.

AVAIL ← pointer to the top element of the availability stack

NEW,SAVE ← Temporary pointer variables

1. [Underflow?]

   If AVAIL = NULL

   Then Write('AVAILABILITY STACK UNDERFLOW')

   Return(FIRST)

2. [Obtain address of next free node]

   NEW ← AVAIL

3. [Remove free node from availability stack] AVAIL ← LINK(AVAIL)
4. [Copy information contents into new node] INFO(NEW) ← X
5. [Is the list empty?]

   If FIRST = NULL

   Then LINK(NEW) ← FIRST
6. [Does the new node precede all others in the list?]

   If INFO(NEW) ≤ INFO(FIRST)

   Then

   LINK(NEW) ←
   FIRST
   Return(NEW)
7. [Initialize temporary pointer]

   SAVE ← FIRST
8. [Search for predecessor of new node]

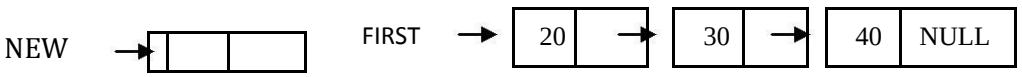
   Repeat while LINK(SAVE) ≠ NULL and INFO(LINK(SAVE))≤ INFO(NEW)

   SAVE ← LINK(SAVE)
9. [Set link fields of new node and its predecessor]

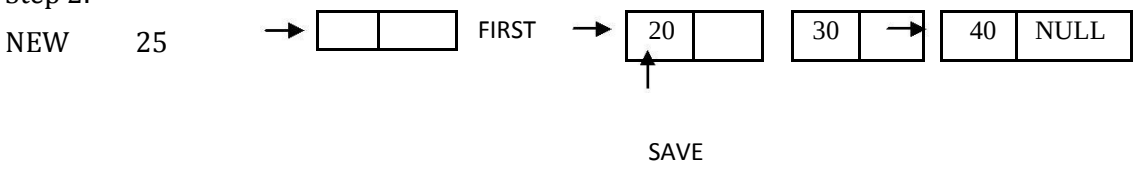
   LINK(NEW) ←
   LINK(SAVE)
   LINK(SAVE) ←
   NEW
10. [Return first node pointer]

    Return(FIRST)

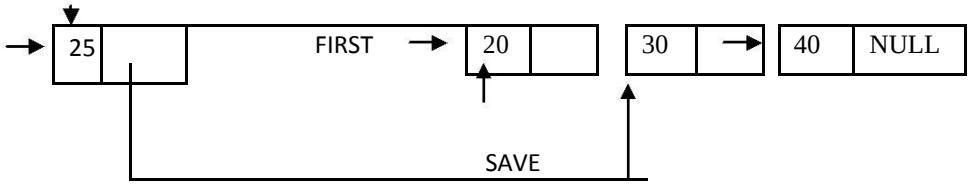
Step 1:




Step 2:




Step 3:




Step 4:


SRI MANAKULA VINAYAGAR ENGINEERING COLLEGE

**Algorithms for inserting an item at the end of the list:**

1.Set space for new

node x 2.Assign value

to the item field of x

3.Set the next field of x

to NULL

4.Set the next field of N2 to point to x

**Function** INSEND(X,FIRST)

**Variables used:**

X ← new element

FIRST ← Pointer to the first element whose node contains INFO and LINK fields.

AVAIL ← pointer to the top element of the availability stack

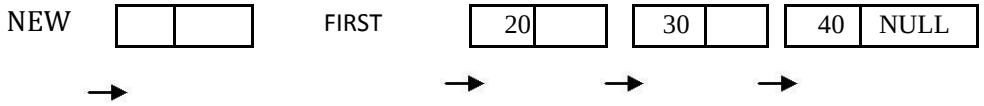
NEW,SAVE ← Temporary pointer variables

1. [Underflow?] If AVAIL = NULL
   Then Write('AVAILABILITY STACK UNDERFLOW')
   Return(FIRST)
2. [Obtain address of next free node]

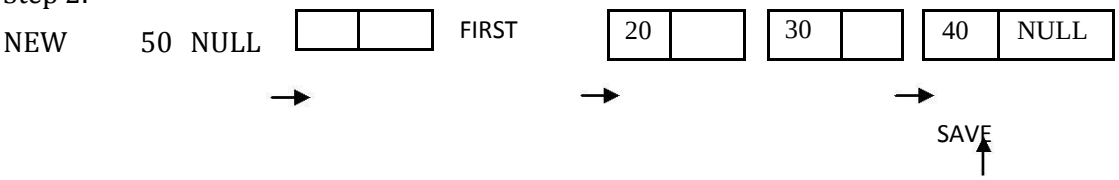
NEW ← AVAIL

3. [Remove free node from availability stack] AVAIL ← LINK(AVAIL)
4. [Initialize fields of new node]
   INFO(NEW) ← X
   LINK(NEW) ← NULL
5. [Is the list empty?]
   If FIRST = NULL
   Then Return(NEW)
6. [Initiate search for the last node] SAVE ← FIRST
7. [Search for end of list]
   Repeat while LINK(SAVE) ≠ NULL
   SAVE ← LINK(SAVE)
8. [Set LINK field of last node to NEW]
   LINK(SAVE) ← NEW
9. [Return first node pointer]
   Return(FIRST)

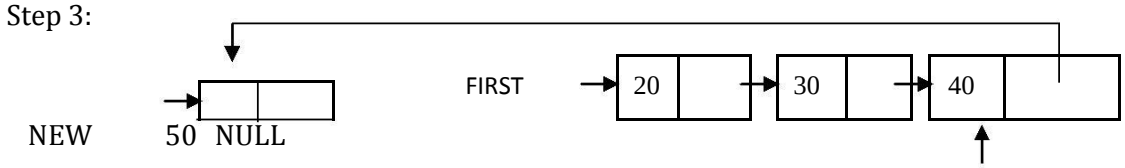
Step 1:




Step 2:




Step 3:




Step 4:


Note that no data is physically moved, as we had seen in array implementation. Only the pointers are readjusted.

**DELETING AN ITEM FROM A LIST:**

Deleting a node from the list requires only one pointer value to be changed, there we have situations: 1.Deleting the first item

2.Deleting the last item

3.Deleting between two nodes in the middle of the list.

**Algorithms for deleting the first item:**

1.If the element x to be deleted is at first store next field of x in some other variable y. 2.Free the space occupied by x

3.Change the head pointer to point to the address in y.

**Algorithm for deleting the last item:**

1.Set the next field of the node previous to the node x which is to be deleted as NULL 2.Free the space occupied by x

**Algorithm for deleting x between two nodes N1 and N2 in the middle of the list:**

1.Set the next field of the node N1 previous to x to point to the successor field N2 of the node x. 2.Free the space occupied by x.

**Procedure** DELETE(X, FIRST)

**Variables used:**

X ← New element to be inserted

FIRST ← Pointer to the first element whose node contains INFO and LINK fields.

TEMP ← To find the desired node

PRED ← keeps track of the predecessor of TEMP

1. [Empty list?]

   If FIRST = NULL

   Then Write('UNDERFLOW")

   Return

2. [Initialize search for X] TEMP ← FIRST
3. [Find X]

   Repeat thru step 5 while TEMP ≠ X and LINK(TEMP) ≠ NULL

4. [Update predecessor marker] PRED ← TEMP
5. [Move to next node]

   TEMP ← LINK(TEMP)

6. [End of the list]

   If TEMP ≠ X

   Then Write('NODE NOT FOUND")

   Return

7. [Delete X]

   If X = FIRST (Is X the first node?)

   Then FIRST ← LINK(FIRST)

   Else LINK(PRED) ← LINK(X)

8. [Return node to availability area] LINK(X) ← AVAIL

   AVAIL ← X

   Return