JAIN DEEMED-TO-BE UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND IT

Master of Computer Applications

Data Structures

(23MCAC101)

Module 1

LIST ADT

Syllabus

List ADT

- Abstract Data Types, List ADT, Static and Dynamic Arrays, Array Operation.
- Linked List Implementation, Singly Linked List, Circularly Linked List,
- Doubly Linked List- all operations–Creation, Insertion, Deletion, Search and Traversal, Circular Linked Lists
- Applications of Linked Lists-Polynomial addition and subtraction – Sparse Matrix Implementation.

Introduction to DS

Definition:

- a **way of organizing the data** in an **effective manner** for providing **efficient data access**

Ex: 1. How kitchen items are organized
2. Organizing Books in Library

Types of DS:

- **Linear** Data Structure
- **Non-Linear** Data Structure

Introduction to Linear DS

– Data are **arranged in a sequential manner**

- Each **data is connected** to its **previous and next element**

Ex:

List – elements are **arranged in a sequence**

Stack - **LIFO (Last In-First Out)** working principle

Queue- **FIFO (First In-First Out)** working principle

Abstract Data Type

- Data type
  - a **set of objects + a set of operations**
  - Example: **integer**
    - set of whole numbers
    - operations: **+, -, x, /**

- **Abstract** data type
  - **High-level abstractions** (managing complexity through abstraction)
  - **Encapsulation**

Abstract Data Type

- Examples
  - the ***Set* ADT**
    - A **set of elements**
    - **Operations**: ***Union, Intersection, Size* and *Complement***
  - the ***Queue* ADT**
    - A **set of sequences of elements**
    - **Operations:** C*reate empty queue, Insert, Search, Delete*, and D*estroy queue*

List ADT

- A sequence of elements arranged in the following order

$$A_1, A_2, A_3, \ldots A_N$$

- N: **length of the list**
- A_1: **first element**
- A_N: **last element**
- A_i: position i
- If **N=0, then empty list**
- Linearly ordered
  - A_i precedes A_{i+1}
  - A_i follows A_{i-1}

Operations

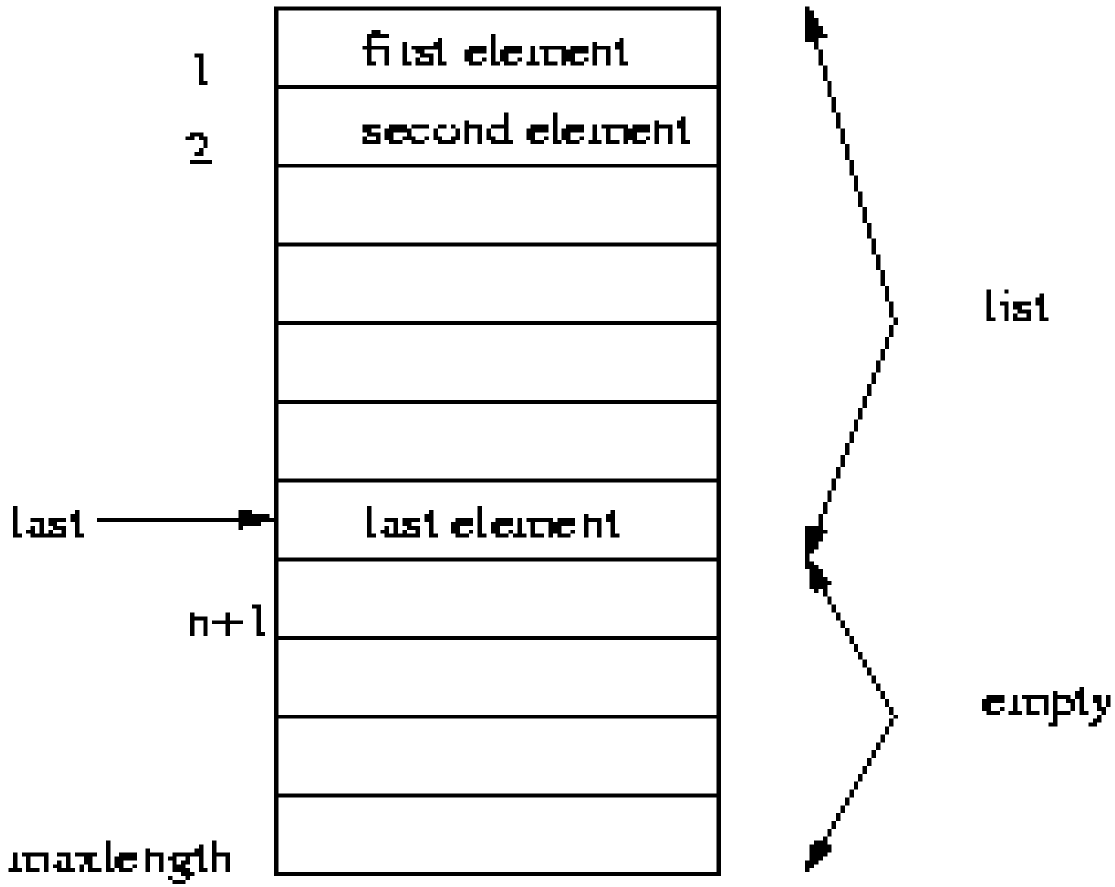
- **makeEmpty:** create an empty list
- **printList:** print the list
- **find:** locate the position of an object in a list
  - list: 34,12, 52, 16, 12
  - find(52) → 3
- **insert:** insert an object to a list
  - insert(x,3) → 34, 12, 52, x, 16, 12
- **remove:** delete an element from the list
  - remove(52) → 34, 12, x, 16, 12
- **findKth:** retrieve the element at a certain position

Implementation

- Choose a **data structure** to **represent the ADT**
  - E.g. Arrays, Records, etc.
- **Each operation** associated with the ADT is implemented by one or more **subroutines**
- Two **standard implementations** for the list ADT
  - **Array-based**
  - **Linked list**

List using Array

Elements are stored in contiguous array positions



Array Implementation

- Requires an estimate of the maximum size of the list
  - waste space
- **printList** and **find**: linear
- **findKth**: constant
- **insert** and **delete**: **slow**
  - e.g. **insert at position 0** (making a new element)
    - requires **first pushing the entire array down** one spot to make room
  - e.g. **delete at position** 0
    - requires shifting all the elements in the list up one
  - On **average**, **half of the lists needs to be moved** for either operation

Static Array

- The **size of the array will be fixed in program code** itself
- The **memory allocation occurs during compile time**.
- The **array size is fixed and cannot be changed.**
- The **location is in Stack Memory Space**.
- **This array can be Initialized but not erased.**

Examples:

1. int a[] = {10,20,30,40,50}
2. int a[10]

Dynamic Array

- The **size of the array will not be fixed in program code**
- The **memory allocation occurs during run time**.
- The **array size can be changed.**
- The **location is in Heap Memory Space**.
- **This array can be erased or deleted from memory**

Examples:

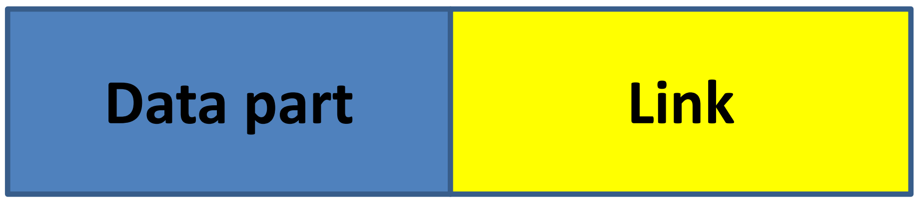
int *array;

array= (int *) **malloc(sizeof(int) * noofelements)) // Dynamic Memory Allocation**

Linked List

Data are stored
- in form of **nodes**
- and those are connected by their addresses in the memory

Node Structure:



Types of Linked List:

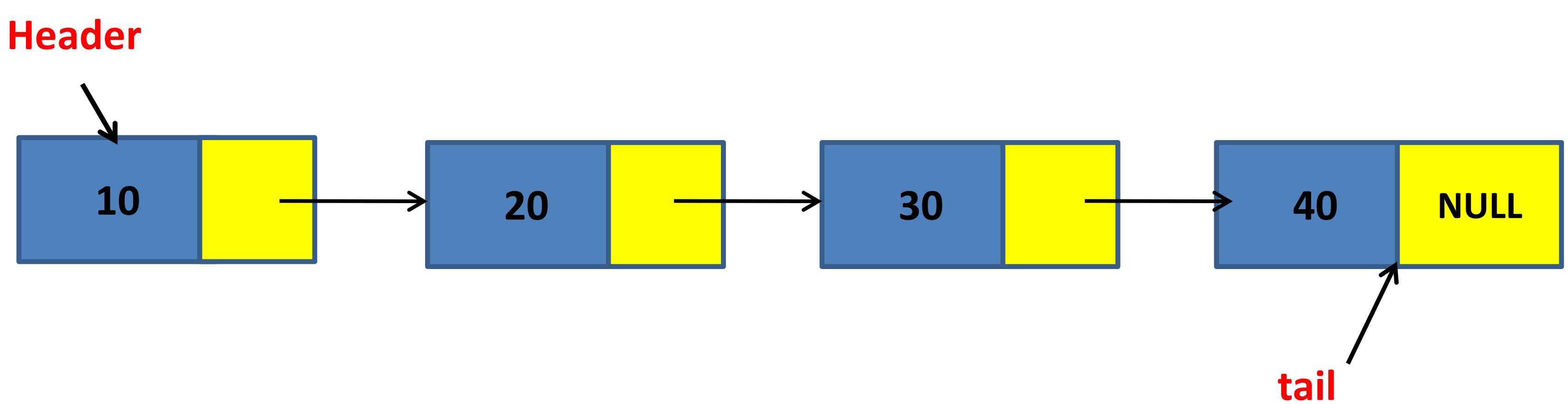
Singly Linked List (SLL)

Doubly Linked List (DLL)

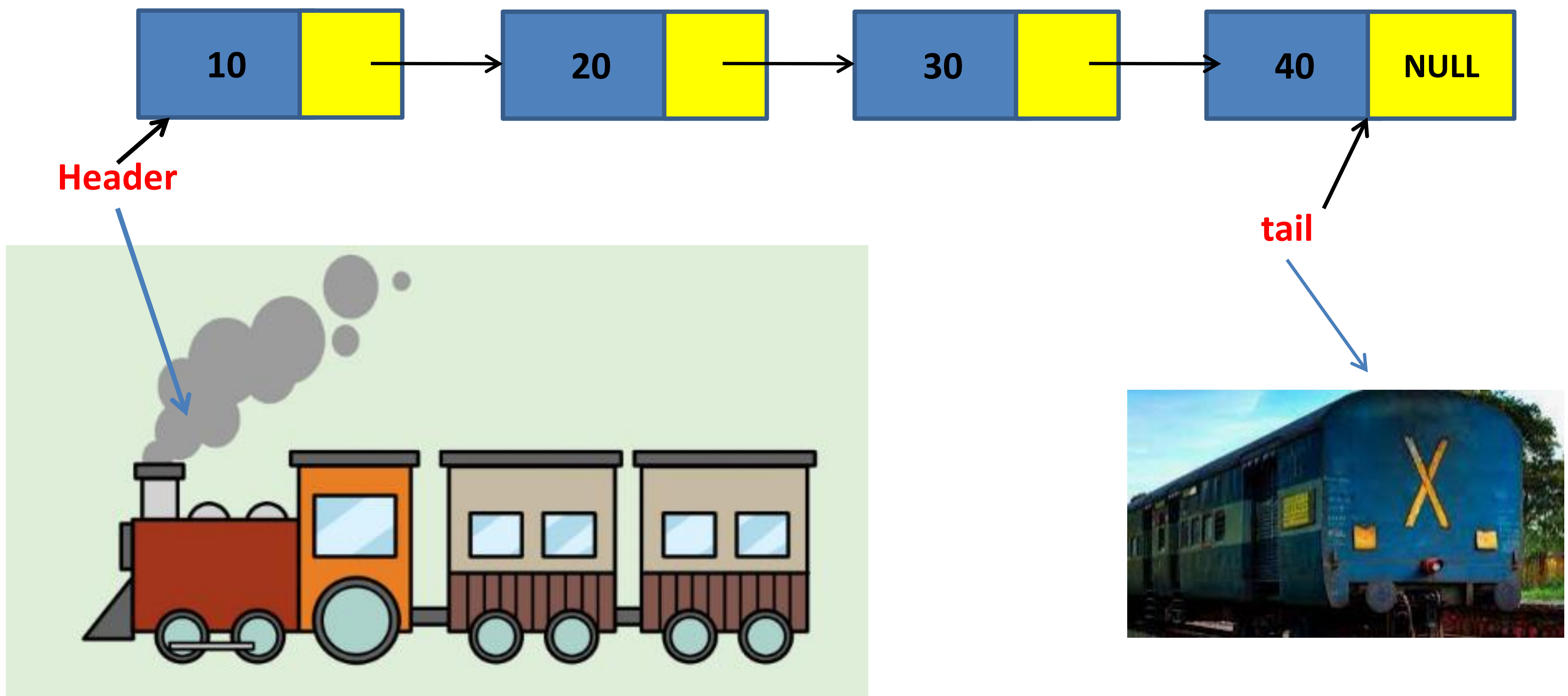
Circular Linked List (CLL)

Singly Linked List

- A *linked list* is a series of connected *nodes*
- Each **node contains** at least
  - A **piece of data** (any type)
  - **Pointer to the next node** in the list
- ***Head***: pointer to the first node
- The **last node points** to **NULL**



Singly Linked List



How to implement

- **Declare Node** for the nodes
  - **data:** Type of data
  - **next:** a pointer to the **next node in the list**

We use structure in C/C++

```
struct Node {
      Type  data;        // data
      Node* link;        // pointer to next
};
```

SCHOOL OF COMPUTER SCIENCE AND IT

Creating an Empty List

- Declare head pointer, which contains
  - head: a **pointer to the first node** in the list.

  Since the **list is empty initially**, head is set to NULL

  So, **Creating an Empty List**

```
struct Node *head = NULL;
```

SLL Operations

- Operations of List
  - **IsEmpty:** determine whether or not the list is empty
  - **InsertNode:** insert a new node at a particular position
  - **FindNode:** find a node with a given value
  - **DeleteNode:** delete a node with a given value
  - **DisplayList:** print all the nodes in the list

1. Insert Node at Front

Inserting a new node at the front of List

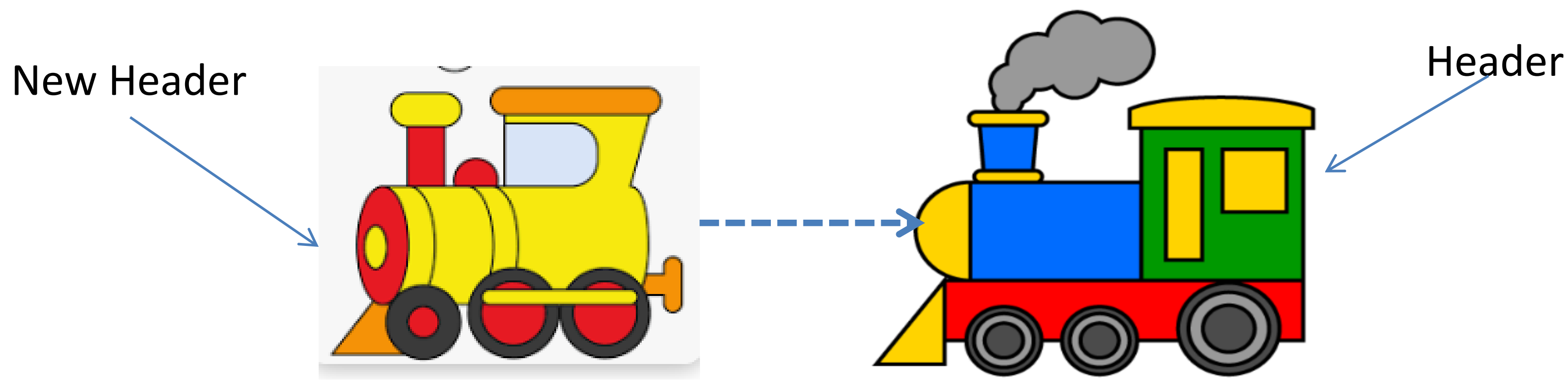
- before the 'Head' node, the given new node will be inserted

- **if List is empty** then the **new node becomes 'Head'** of the list

Else

'link' in new node = 'address' of 'Head' node and

'Head' pointer is **jumped to new node (Head = newNode)**



Code/Pseudocode

```
InsertFront() {
        n= (struct node*) malloc(sizeof(struct node));
        printf("ENTER THE DATA : ");
        scanf("%d",&n->data);

    if(head==NULL) {
                n->link = NULL;
                head = n;
    }

    else {
                n->link = head;
                head = n;
      }

}
```

In empty list, the new node becomes head of the list

Before the head node, the new node is attached and it becomes head of the list

2. Inserting Node at End

SCHOOL OF COMPUTER SCIENCE AND IT

Inserting a new node at the End of the List

if List is empty then

the **new node becomes 'Head**' of the list

Else

- **Reach the last node** in the list by doing traverse the list
- then **assign the address of new node** in **'next' link of last node**

Note:

- new node **'next' link part** should have **'Null' value** to state the end of list

Code/Pseudocode

InsertLast() {

```
struct node *temp,*r;
r=(struct node*) malloc(sizeof(struct node));
printf("ENTER THE DATA : ");
scanf("%d",&r->data);
r->next = NULL;
        temp=head;
        while(temp->next!=NULL)
                temp=temp->next;
        temp->next = r;
}
```

Traversing the list to reach last node then attach new node at the end of the list

In Middle of the List

Inserting a new node in the middle of the list

- **'index' or location** where new node to be inserted
- **traverse the list** to reach the **'index' position**
- **'next' link in new node** will **get the address of the next node of 'temp'** which indicates the given 'index'
- **assign the new node address** to the **"next" field of temp node**
- if **no sufficient nodes** in the list to reach the given index then **error will be generated**

Code/Pseudocode

```
InsertMiddle() {
        \\ read 'loc' to where new node to be inserted
        temp=head;
      for(i=0;i<loc-1;i++) {
          temp=temp->next;
          if(temp==NULL) {
            printf("LESS NO. OF NODES\n");
        } }

        r->next = temp->next;
        temp->next = r

  }
```

Moving temp pointer to the required location

New node r is added after the given 'loc' index in the list

Finding a Node

- void FindNode(type x)
  - **Search for a node** with the **value equal to x** in the list.
  - If **such a node is found**, **return its position**. Otherwise, return 0.

```
void FindNode(int x) {
        temp = head; i=0;
        while(temp!=NULL) {
            if(temp->data == x){
               //display the location
               return 0;
          }
          else{
               temp=temp->next;
               i++;
            }
        }
        //display that "item not found";
      }
```

Finding key present in the list

Deleting a Node

DeleteNode(type x)

- **Delete a node with the value equal to x** from the list.
- Release the node from the list.

- **Steps**
  - Find the desirable node (similar to FindNode)
  - **Set the pointer of the predecessor to the successor of the found node**
  - **Release the memory occupied** by the found node

Contd..


```
void DelItem() {
      .........
    temp = pre = head;
    while(temp!=NULL) {
       if(temp->data==x){
          if(temp==head){ head=temp->next; }
          else { pre->next = temp->next;   }
           free(temp);
           printf("SUCCESSFULLY REMOVED\n");
           return;
       }
      else{              pre = temp;
                         temp=temp->next;
           }
    }
  }
```

X found, and check it is in head node or intermediate node

X not found, jump to next nodes

Print the List

- **DisplayList()**
  - Traverse the list from head using a dummy pointer ‘temp’
  - Print the data of all the elements

```
void DisplayList()
{
    temp = head;
    if(temp ==NULL){
                printf("LIST IS EMPTY\n");
                return;        }
    while(temp!=NULL){
                printf("%d\t",t->data);
                temp=temp->next;        }
}
```

Display data of each node in the list

Destroy the List

- **Steps**
  - **Start from head** and jump to next nodes
  - Use the **'free()' to release** the memory of nodes one by one
  - Finally, make **'head=NULL'**.

```
void DestroyList() {
        temp = head;
        while(temp!=NULL){
            r = temp;
            temp= temp->next;
            free(r);
        }
    head=NULL;
}
```

Release each node from the list

DLL

- *Doubly linked lists*
  - **Each node points** both **successor and the predecessor**
  - There are **two NULL**: at the **first and last nodes** in the list
  - Advantage: given a node, it is **easy to visit its predecessor**.

    > Convenient to **traverse lists backwards**

DLL

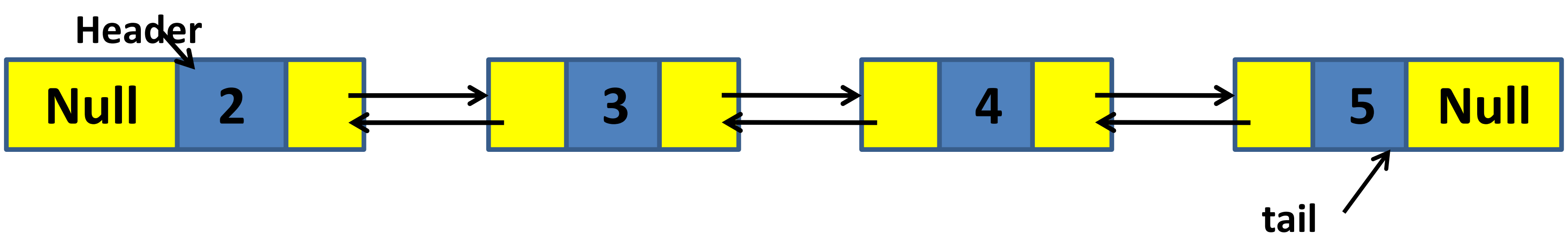
In DLL

- a Node having two Link parts

prev – to connect **previous node**

next – to connect **next node**

Node Structure:



SCHOOL OF COMPUTER SCIENCE AND IT

- **Declare Node**
  - **data**: Type of data
  - **next**: a pointer to the next node of the current node
  - **prev**: pointer the previous node of the current node

```
struct DNode {

      Type  data;            // data
      Node *next, *prev;    // pointer to next & prev nodes
};
```

Insert Node at Front

Inserting a new node at the **front of List**

- before the **'Head'** node the given new node will be inserted
- **if List is empty** then the **new node becomes 'Head'** of the list

else

- **'next' link in new node** will **get 'address' of 'Head' node** and
- **'prev' field in 'Head'** node will **get address of new node**
- **move 'Head'** pointer **to new node**

Note:

- **'prev' field** in new node **should be 'Null'**

Code/Pseudocode

```
InsertFront() {
        ..... New node creation......
      n ->prev = NULL;

       if(head==NULL) {
                n->next = NULL;
                head = n;
      }

   else {
                n->next = head;
                head = n;
         }

  }
```

In empty list, the new node becomes head of the list

Before the head node, the new node is attached and it becomes head of the list

Inserting Node at End

Inserting a new node at the End of the List

- **if List is empty** then the **new node becomes 'Head'** of the list

else

-**Reach the last node** in the list by traversing the list

-then **copy the address** of **new node** in **last node's next field**

-and **'prev' field of new node** will **get last node address**

Note:

- **new node 'next' link** part **should have 'Null'** value to state the end of list

Code/Pseudocode

```
InsertLast() {
  struct node *temp,*r;
  r=(struct node*) malloc(sizeof(struct node));
  \\read data for new node (r)
  r->next = NULL;
          temp=head;
          while(temp->next!=NULL)
                  temp=temp->next;
          temp->next = r;
          r->prev = temp;
  }
```

Traversing the list to reach last node then attach new node at the end of the list

In Middle of the List

JGi JAIN DEEMED-TO-BE UNIVERSITY SCHOOL OF COMPUTER SCIENCE AND IT

Attaching a new node in the middle of the list

- **'index'** has to be assigned where it has to be inserted
- traverse the list **upto the 'index' position**
- if **no sufficient nodes** to reach the given index then **error will be arise**
- After reaching the position, following procedure to be done

new->next = temp

new->prev = temp->prev

temp->prev->next = new

temp->prev = new

```
InsertMiddle() {
        \\ read 'loc' to where add new node
        temp=head;
    for(i=0;i<loc;i++) {
          temp=temp->next;
          if(temp==NULL)
                printf("LESS NO. OF NODES\n");
      }
    r->next = temp;
    r->prev = temp->prev
    temp->prev->next = r
    temp->prev = r          }
```

New node r is added after the given 'loc' index in the list

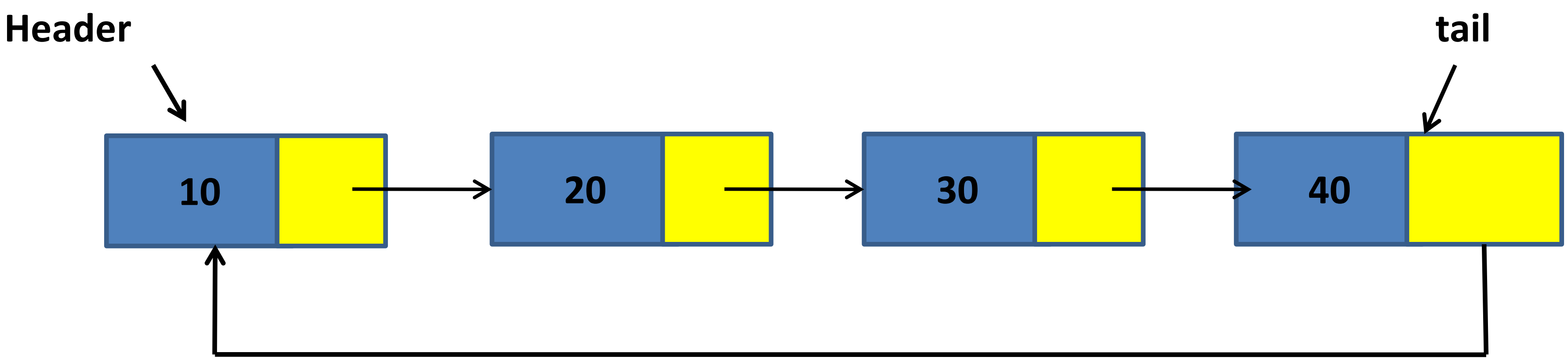
Print the List

- **DisplayList()**
  - Print the data of all the elements
  - Print the number of nodes in the list

```
void DisplayList()
{
    t =start; non=0;
    if(t==NULL){
              printf("LIST IS EMPTY\n");
              return;       }
    while(t!=NULL){
            printf("%d\t",t->data);
            t=t->next;
             non++;  }
}
```

CLL

- *Circular linked lists*
  – The last node points to the first node in the list



– How do we know when we have finished traversing the list?

(Tip: check if the pointer of the current node is equal to the head.)

CLL

- *2 ways to implement*
  - **Using Singly Linked List**

    - the **'next'** field in **last node** should **have address of 'Head**'

  - Using **Doubly Linked List**

    -The **'next'** field in last node should **have address of 'Head'**

    - **'prev' field in 'Head'** node will have the **address of last node**

Arrays VS LL

- **Linked lists are more complex to code** and **manage than arrays**, but they have **some distinct advantages**.
  - **Dynamic**: a linked list **can easily grow** and **shrink in size**.
    - **We don’t need to know how many nodes** will be in the list. They are created in memory as needed.
    - In contrast, **the size of a C++ array** is **fixed at compilation time**.
  - **Easy and fast insertions and deletions**
    - To **insert or delete an element in an array**, we need to **copy to temporary variables** to **make room for new elements** or close the gap caused by deleted elements.
    - With a linked list, **no need to move other nodes**. Only need **to reset some pointers**.

Applications of Linked List

- **Polynomial Manipulation representation**
- **Representation of sparse matrices.**
- Addition of long positive integers.
- Symbol table creation.
- Mailing list.
- Memory management.
- Linked allocation of files.

Polynomial Manipulation

- Given two polynomial numbers represented by a linked list.
- To add these lists means add the coefficients who have same variable powers.

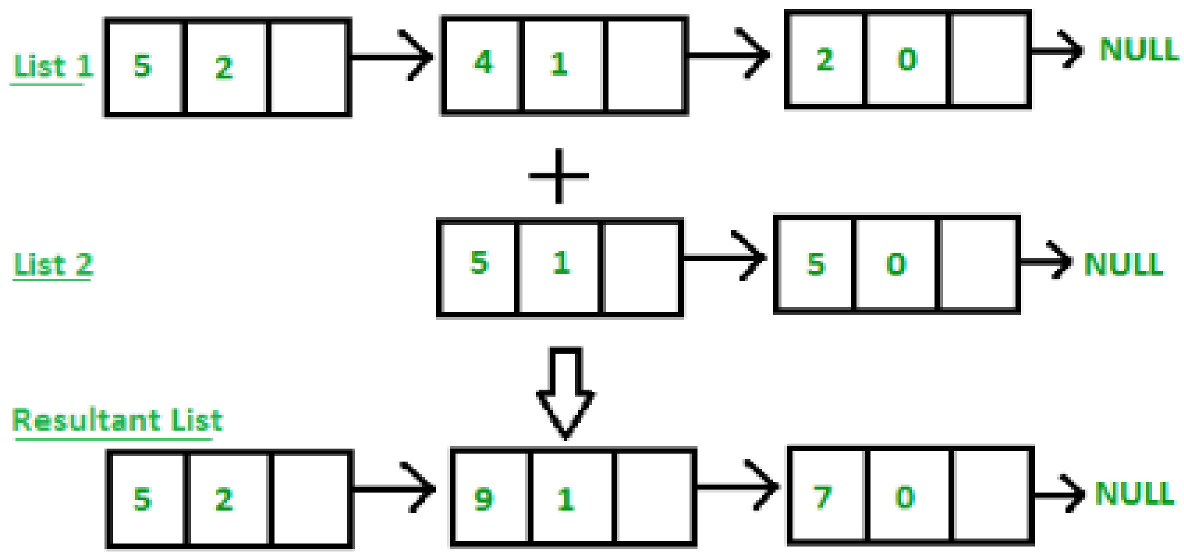
Input:

1st number $= 5x^2 + 4x^1 + 2x^0$

2nd number $= -5x^1 - 5x^0$

Output:

$5x^2 - 1x^1 - 3x^0$



Sparse Matrix

- A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values.
- If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.

Why to use Sparse Matrix instead of simple matrix ?

- **Storage:** There **are lesser non-zero elements than zeros** and thus lesser memory can be used to store only those elements.
- **Computing time: Computing time can be saved** by **logically designing a data structure traversing only non-zero elements**..

Sparse Matrix

Array representation of the sparse matrix

0	0	3	0	4
0	0	5	7	0
0	0	0	0	0
0	2	6	0	0

- Representing a sparse matrix by a **2D array leads to wastage of lots of memory as 0s** in the matrix are of no use in most of the cases.

2D array is used to represent a sparse matrix in which there are 3 rows named as

Row: Index of row, where **non-zero element** is located

Column: Index of column, where non-zero element is located

Value: Value of the non zero element located at index – (row, column)

Sparse Matrix

Array representation of the sparse matrix

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Sparse Matrix

Linked List representation of the sparse matrix

In linked list, **each node has four fields**.

These four fields are defined as:

Row: Index of row, where non-zero element is located

Column: Index of column, where non-zero element is located

Value: Value of the non zero element located at index – (row,column)

Next node: Address of the next node

Sparse Matrix

Linked List representation of the sparse matrix